Summer 8-12-2016

Distributed Graph Storage And Querying System

Janani Balaji

DISTRIBUTED GRAPH STORAGE AND QUERYING SYSTEM

by

JANANI BALAJI

Under the Direction of Rajshekhar Sunderraman, PhD

ABSTRACT

Graph databases offer an efficient way to store and access inter-connected data. However, to query large graphs that no longer fit in memory, it becomes necessary to make multiple trips to the storage device to filter and gather data based on the query. But I/O accesses are expensive operations and immensely slow down query response time and prevent us from fully exploiting the graph specific benefits that graph databases offer.

The storage models of most existing graph database systems view graphs as indivisible structures and hence do not allow a hierarchical layering of the graph. This adversely affects

query performance for large graphs as there is no way to filter the graph on a higher level without actually accessing the entire information from the disk. Distributing the storage and processing is one way to extract better performance. But current distributed solutions to this problem are not entirely effective, again due to the indivisible representation of graphs adopted in the storage format. This causes unnecessary latency due to increased inter-processor communication.

In this dissertation, we propose an optimized distributed graph storage system for scalable and faster querying of big graph data. We start with our unique physical storage model, in which the graph is decomposed into three different levels of abstraction, each with a different storage hierarchy. We use a hybrid storage model to store the most critical component and restrict the I/O trips to only when absolutely necessary. This lets us actively make use of multi-level filters while querying, without the need of comprehensive indexes. Our results show that our system outperforms established graph databases for several class of queries. We show that this separation also eases the difficulties in distributing graph data and go on propose a more efficient distributed model for querying general purpose graph data using the Spark framework.

INDEX WORDS: Graph Databases, Distributed Graph Databases, Distributed Graph Query Processing, Spark

DISTRIBUTED GRAPH STORAGE AND QUERYING SYSTEM

by

Janani Balaji

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2016

DISTRIBUTED GRAPH STORAGE AND QUERYING SYSTEM

by

JANANI BALAJI

Committee Chair: Rajshekhar Sunderraman

Committee: Yinghsu Li

Yanqing Zhang

Hendricus Van der Holst

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

August 2016

ACKNOWLEDGEMENTS

This dissertation work would not have been possible without the support of many people. Primarily, I want to express my gratitude to my advisor Dr. Rajshekar Sunderraman, for believing in me and patiently guiding me towards this milestone with his knowledge and expertise. I would like to thank him for empowering me to grow as a computer science researcher. I also express my thanks to each of my committee members - Dr. Yingshu Li, Dr. Yanqing Zhang, Dr. Hendricus Van der Holst, for their continuous support and encouragement. I would like to thank all the administrative staff at the Department of Computer Science for efficiently taking care of all the tedious organizational tasks behind the scene. I also take this opportunity to thank Dr. Carol Winkler, Dr. Tony Lemieux, Dr. Saeid Belkasim and the entire Transcultural Conflict and Violence (TCV) team for supporting me through the last four years and providing me an opportunity to utilize my skills in an inter-disciplinary environment. The many discussions we had, helped me expand my perspectives on outside my field of study.

A special note of thanks to all my professors who contributed to my growth, both personally and professionally, with their positive influences. Finally, I take this opportunity to thank all my fellow students, who made my last five years more enjoyable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PART 1

INTRODUCTION

Usage and generation of interconnected data is on the rise. Be it Social Networks, Biological Networks, Internet of Things or Business analysis, there is a growing emphasis on relating and inferring information from different entities. Be it a Social Network, that describes the characteristics of a person as well as the characteristics of other people he is connected to or a Biological Network, in which the interactions and effects between different chemicals are represented as an interconnected network, there is a marked increase in the generation and consumption of such interrelated data. The number of Facebook users grew from 500 million in 2010 to 1.5 billion in 2014 [1]. The Common Crawl web corpora [2] covered for 2012 contains 3.5 billion web pages and 128 billion hyperlinks between these modeled as a graph. This growth describes the increasing usage of inter-related data in all fields. As more and more such data is generated, it becomes imperative to devise specialized storage and retrieval mechanisms for optimal performance, both in time and scalability.

Graphs are ideal structures for representing such interrelated data. The vertices in a graph represent the objects of interest and the edges represent the relationships between the objects. Hence, graphs provide a visual representation that closely represents the actual data of interest. Graphs started out as a simple structure consisting of a set of vertices and their connecting edges and have now developed several variations including the property graphs in which the edges and vertices can have attributes of their own, hypergraphs where there can be more than one edge between a pair of vertices, trees a restricted form of graphs that have no cycles, etc.,. Graphs are also semi-structured in nature, where, there is generally no fixed schema that it needs to confirm to. The attributes that each vertex/edge can have is purely optional without a need to adhere to fixed restrictions. The vertices and edges in a graph generally belong to specific categories, known as types. For example, in a Social

Networking environment, the different vertex types could be “Person”, “Place”, “School” whereas the different edge types can be “Knows”, “Lives in” etc. depending on how the network is modeled.

1.1 Graph Model

The class of graphs we are dealing with are called Property Graphs, in which both vertices as well as edges can have properties of their own. Vertices and edges belong to one of the predefined types and there can be more than one edge between two vertices. The edges can be directed or undirected. These are the most generalized forms of graphs and are frequently found in real time scenarios like Social Media, Traffic Analysis, and Business Analytics etc. A typical property graph model is given in Figure 1. Each vertex and edge has a unique id, which is enclosed within the circle for vertices and specified on itself for edges. This model is semi-structured in that both vertices and edges can choose to have their own subset of properties from the entire property domain. More formally, a network G of related entities is modeled as a graph $G = \{V, E, \Sigma, \Phi\}$, where V is the set of vertices, $E \in VXV$, is a set of edges, Σ is a set of vertex labels and Φ is a set of edge labels.

1.2 Why Graph Databases?

The key challenge in graph-data storage arises from the need to optimize the storage structure for both the actual data and the relationships. Traditional relational databases are ill-suited for storing such graph data. Consider the graph shown in Figure 1.1 representing a social network interaction. Each vertex and edge is named, and belongs to a specific type, with properties of its own.

In order to represent this graph information in a typical normalized relational database, we would have at least five different tables one to store vertex types, one to store edge types, another to store the edges and one table each for the vertex and edge properties. Now consider a typical graph query to this social network. We might be interested in finding all people who know a person working for the company “ABC Corp”. If we try to execute

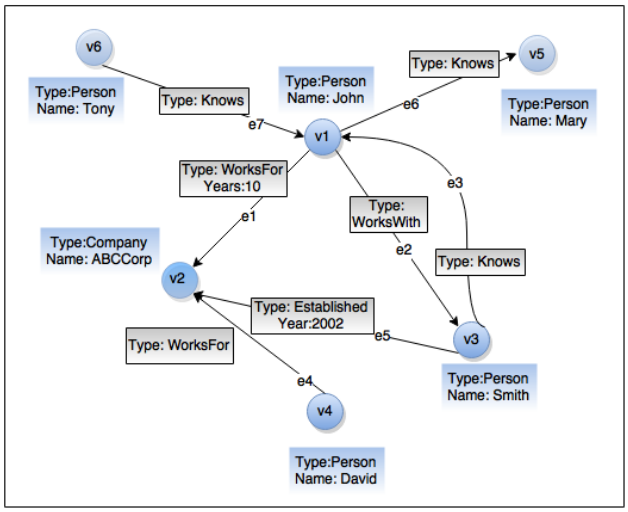


Figure (1.1) Sample Graph

this query in a relational database, there are at least 2 joins involved. One to extract people working for "ABC Corp" and another to extract the friends. Typically, for a query with "n" relationships, "n-1" joins are required in a relational database. But, joins are notoriously expensive operations that severely limit query performance. Also, the ordering of joins plays a significant role in the performance of a relational database. If not efficiently optimized and ordered, a join operation might involve entire tables in the database, leading to significant increases in query time. Though indexing and query processing methods have been tremendously researched and improved, as the complexity of the graph query increases, the querying methodologies of relational databases become both cumbersome as well as inefficient.

The other major drawback in using a tabular structure for storing graph-like data is the difficulty in storing properties. Graph data is by nature semi-structured, in the sense, it does not strictly adhere to a pre-defined schema. A relational database is ideally suited for data that conforms to a standard layout, so that the number of columns in a table can be pre-determined. But consider a typical graph data in a social networking setting. The attributes can vary widely within the objects of the same graph. The presence/absence

of an attribute is optional for any vertex or edge, unless explicitly specified. Hence, it is impossible to determine the number of attributes the object can have at design time. This leaves us with two alternatives; one - to use a de-normalized approach to store properties and add additional columns whenever a new property is added to the system, and the other - to completely normalize the database and add a new table for every attribute created.

Both systems have drawbacks of their own. The de-normalized approach does not fit well because, due to the irregularity in the properties, we end up having a sparse matrix, leading to wasted storage space. Also, data modifications become cumbersome, as adding/deleting columns is a data definition operation and involves modifying the entire table. This adds significant time for large datasets. The normalized approach on the other hand uses a separate table for each property. This results in creating a new table every time a property is added and also results in an additional layer of join operation in the query. The normalized model also results in multiple lookups to retrieve all the properties of a given vertex/edge, the very basic operation from a graph perspective.

On the other hand, from a graph perspective, this query only involves locating the vertex representing "ABC Corp" and then traversing the graph for a depth of two one to find the people who "work for" the corporation and the next to find people who are connected to them with a "knows" edge. In essence, after locating a starting point, the region of interest is contained to a very small radius, making the query search space manageable. A graph database aims to bring in this benefit to querying relationships. In a graph database, data storage is optimized for graph operations and hence does not suffer from drawbacks of a relational database. The benefits are especially amplified in a large data setting, where this reduction in search space directly translates into an increase in performance.

In a graph database, most queries are executed using a graph traversal operation, a basic graph operation which is extensively studied and optimized. It is imperative to examine the structure of a typical graph query at this point. Let us consider the most typical of graph queries, the path query. A path query is one which returns a set of vertices and edges that confirm to certain restrictions. The restrictions can be in the form of attributes of

the vertex/edge or about the structure by which the set of vertices and edges need to be connected. We can generalize a path query as shown in Figure 1.2.

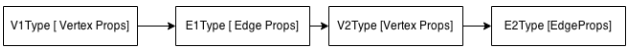
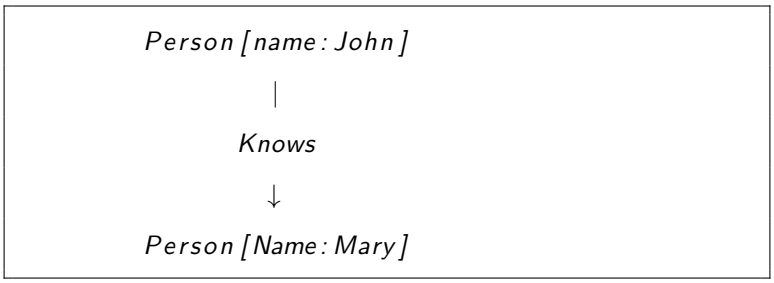


Figure (1.2) Path Query

A path query can be constructed using repeated blocks of VType and EType, each having their own optional properties. An example would be



This is a path query that filters all vertex objects of type *Person* and with name *John* who are connected by an edge of type *Knows* to a vertex object of type *Person* and name *Mary*. As we observe, a graph query is as much about the data contained in the graph as it is about the structure. While a relational database separates the structure from the data, a graph database provides a unified structure that addresses both the data as well as the structure as one whole unit.

A graph database is also whiteboard-friendly, in the sense that the logical model of data is very close to the visual representation. This enables us to directly translate ideas from the visual domain to the logical domain without the need for intermediate data translations. Also, a purely graph-like representation lets us make use of the wealth of graph algorithms that have been developed. Graph theory is a very old field of mathematics and lots of research has been made in this area. Several graph algorithms are available for operations like clustering, shortest-path, centrality measures etc., which directly correspond to several inference and analytic operations. A graph database serves as an ideal medium that abstracts the physical variations in storage and lets us focus on the higher level analytical details.

1.3 Kinds of Big Graphs

When we talk about big graph data, they come in two flavors. The first kind are the transactional graphs, which are a large collection of small graphs, typically with > 50 vertices. A typical transactional graph collection can have millions of such graphs. These kind of graphs are generally found in bio-informatics and chemical studies to name a few, where each of these graphs can be used to represent a specific protein structure or a chemical compound, drug interactions, etc., The other kind of graphs in the big data world are those single large graphs that contains billions of vertices and edges to form one large graph. These graphs are typically used to model social network sites, internet-of-things data, networking, business analysis and so on. Though both are graphs in general and follow similar structures, there are wide differences in their usage and characteristics that affect the storage and querying mechanisms. The model we develop work well for both transactional as well as single large graphs, though at present we concentrate on the **single large graph** case.

1.4 Graph Processing

While graphs are ideal representations for connected data, their significance does not end with being illustration-friendly structures. There is a wealth of information that can be gleaned by adopting a graphical representation. Graph theory is a very well developed branch of mathematics with several established methods to estimate and dissect the nature of network data. The PageRank is a well-known algorithm that uses the "Random Walk" technique of graphs to determine the importance of a web page. The "Connected Components" technique has been used in multiple fields including Entity Resolution to identify clusters of entities. Understanding the diameter, flow-rate, density, degree etc.,. have their use in constructing reliable, robust network architectures.

The vertex coloring problem is a typical solution to scheduling and assignment dilemmas. Centrality finding algorithms like [3], [4], Girvan-Newman [5] are typical graph algorithms that find their use in several community detection applications. Modeling data as a bipartite

network finds its use in recommender systems, a tree-based format is useful in hierarchical data representation. Graph traversal also has several use cases such as edit-distance, sorting, parentheses matching etc.,. Hence, modeling data as a graph lets us tap into the vast collection of graph algorithms and theorems that find practical uses with inter-related data.

1.5 Path Queries

In this dissertation, we focus on simple graph path queries of the type shown in Figure 1.1. Such path queries find applications in several areas including Internet of Things (IoT) systems and Social Networks. Consider the sample graph G and the query graph Q shown in Figure 1.3. It could be used to represent the friendship structure of users in a Social Network or could also be used to represent the running route of a user in an Internet of Things system, depending on how the vertices and edges are modeled.

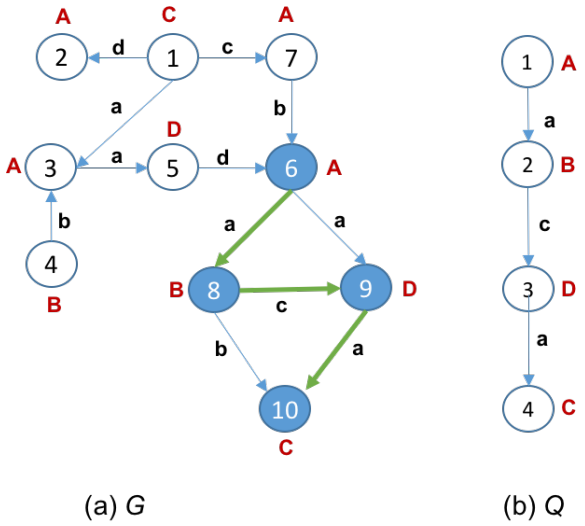


Figure (1.3) Sample Graph G and Path Query Q

We define a path query as consisting of a set of blocks, each block defined by a vertex

label, direction and edge label. A query Q of length n can be defined as

$$X = \begin{cases} (s_i, d_i, l_i), & \text{if } i = 0, 1, ...n-2 \\ s_i, & \text{if } i = n-1 \end{cases} \tag{1.1}$$

where $s_i \in \Sigma$ is the vertex label, $d_i \in \{o, i\}$ is the edge direction and $l_i \in \Phi$ is the edge label. For brevity, we represent each query block as q_i.

For example, the path in Figure 3(b) can be divided into 4 blocks as follows:

block 0 - (A,o,a)

block 1 - (B,o,c)

block 2 - (D,o,a)

block 3 - (C)

Notice that block 3 is defined only in terms of the vertex label as it forms the last block of the path query.

1.6 Distributed Path Queries

A graph is considered to be distributed across the processors, if the data is split into several partitions, with each of the partitions stored in multiple processing units. The major difficulty in partitioning the graph results from the interconnected structure of the graph. For optimal distributions, it is desirable to have fewer edges that crossover between the partitions, so that the information contained in a partition is maximal with respect to answer queries related to the data contained in it.

The partitioning strategy used dictates the performance of the distributed graph application, as it is crucial to maintain the computation to communication balance. While a single hash partitioning, where the vertices are distributed based on the hash-index of their vertex is sufficient, typically, graph partitioning algorithms are utilized to form optimal segments of the graph that minimize edge crossovers between processing nodes.

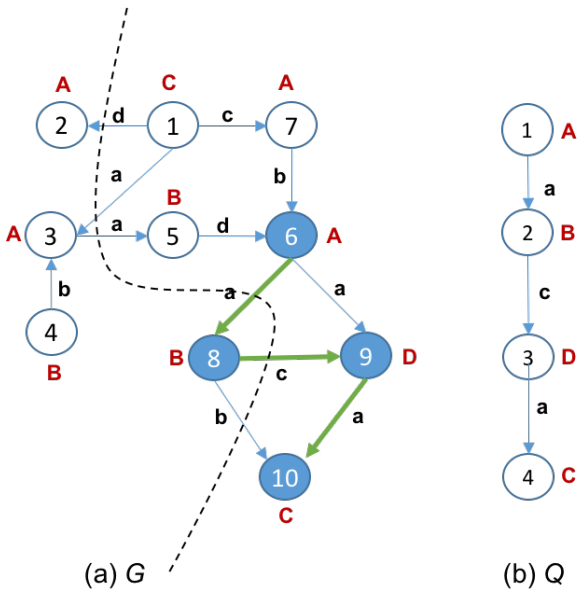


Figure (1.4) Path Query

DEFINITION 1 ***(DISTRIBUTED PATH QUERY)*** *Given a network G distributed over a cluster, and a query path Q, the distributed graph path query problem is to find all the distinct matchings of Q in G, that may or may not span multiple processing nodes.*

EXAMPLE 1 *Figure 1.4(a) represents a network graph that is stored across a cluster, with the partitioning indicated by dotted lines. Figure 1.4(b) represents a query graph. Numerical identifiers are used to represent vertex ids and the vertex labeling is indicated by capital alphabets and edge labeling by small alphabets. The subgraph with $V(G') = \{6, 8, 9, 10\}$, highlighted in blue, matches Q and is hence returned as a matching path.*

1.7 Applications

Graphs find their uses in a variety of fields. Social Networks, Internet of Things, Bio-Chemical networks, Network analysis, Recommendation Systems etc.,. are a few of those. Since in this dissertation, we focus on graph exploration using path queries, in this section we site some examples for graph path queries in two different application fields.

1.7.1 In an Internet of Things system

Consider the network graph representing a users running behavior shown in Figure 1.3(a). Let us assume the data is gathered from the wearable fitness tracker used by the user. Let the vertices represent specific landmarks along the running route and the edges represent the path taken to reach one landmark from the other. Let the vertex labels, denoted by capital alphabets represent the *elevation range* at each landmark and let the edge labels, represented by small letters represent the *average speed* of the user to run along that edge. The legend in Figure 1.5 gives the description of the vertex and edge label types. Representing this data in a graph form gives a visual representation of the data and also lets us uncover relationships through efficient graph traversals. Here, the query in Figure 1.3(b)

Vertex Label	Elevation Range
A	-1
B	0
C	1
D	2

Edge Label	Speed Range (miles/hr)
a	0-2
b	2-3
c	3-4
d	4-6

Figure (1.5) Graph in Figure 1.3 as an IoT system

helps identify frequent running patterns of the user with respect to the elevation of the place and the running speed. Such queries may be used to find the running routes that maximize a person's performance and suggest similar alternate routes.

1.7.2 In a Social Network

The graph and the query path in Figure 1.3 can also be used to represent a Social network, based on how the graph is modeled. Let the graph G represent a social network depicting interactions between people with different occupations. Let the vertex labels denote

occupations and edge labels denote the *relationships* between the people. The legend table in Figure 1.6 gives the descriptions of the vertex and edge labels.

Vertex Label	Occupation
A	Engineer
B	Doctor
C	Teacher
D	Driver

Edge Label	Relationship
a	Knows
b	Student of
c	Treats
d	Drives

Figure (1.6) Graph in Figure 1.3 as a Social Network system

In this context, the query graph in Figure 1.3(b) traces "*All Engineers who know a Doctor who treat a driver who knows a Teacher*".

Such paths are also applicable in identifying network traces, user behavior in a smart environment, traffic cause-effect analysis etc., We represent path queries to start and end with a vertex type, connected in-between by a sequence of alternating edge and vertex types. A path of length thus contains different segments. Though we do not use the syntax for regular path queries, our model can be easily extended to suit them.

Graph path querying involves identifying all matching graph substructures that satisfy the issued query. Consider the graph shown in Figure 1.3(a) and the query given in Figure 1.3(b). The graph substructure matching the query is highlighted in blue in Figure 1.3(a). Since the entire graph data can be huge, in order to restrict the search space, a few seed vertices are identified and the path is grown by following the conforming edges starting from the seed vertices. The seed vertices restrict the search space to a few likely points on the graph and the path is grown either by a traversal operation or by a series of join operations, depending on the data representation format. In either case, for a query of length n there are at least $n-1$ iterations of examining if a certain path is viable and then growing the feasible path.

PART 2

RELATED RESEARCH

The origin of graphs as a significant data structure can be traced to the days of Euler. He famously demonstrated a route through the city by visiting each of the bridges only once [6], by modeling the bridges and the route connecting them as vertices and edges of a graph. However, after a period of inactiveness, graphs started gaining prominence with the development of World Wide Web in the early 80s-90s. With more and more data being collected, people started analyzing this data to predict patterns and infer knowledge and more work was done towards graph data analysis. Several algorithms were developed for graph analytics such as mining frequent sub graphs [7] [8], graph clustering [9], [10] and similarity searches [11] . Most of these were in-memory algorithms that operated on small datasets. Persistence of graph data was not generally required as these algorithms were typically run on the intermediate result of a larger analysis engine.

2.1 Graph Representations

As the size of data increased and graph-like data continued to gain importance, persistence of graph data became necessary. Data storage formats started out from simple XML files to traditional relational databases. This led to research about the storage representations for graph data. The simplest form to store graph information was through adjacency lists and matrices.

An adjacency list stores the unique ids of all the connecting vertices for each vertex in the graph in the form of a list. This was a simple enough data structure to store the graph information as it captured the essence of the graph by storing the neighborhood of each vertex as a list and also accounted for easy graph inserts, deletions and modifications. However, it could not handle properties for both vertices as well as edges. An adjacency

matrix on the other hand was more inclusive in that, it allowed us to store weighted graphs. An adjacency matrix is a square matrix whose size was equal to the total number of vertices in the graph. The presence or absence of an edge between two given vertices was represented by a non-zero entity in the matrix position given by the corresponding vertices. However, it was not possible to store multiple properties for the graph objects and also, the size of the matrix was a prohibitive factor, especially for sparse matrices. Both these forms of representation required additional storage mechanisms to store vertex properties. The adjacency list/matrix representation was merely an index system built on the original information about the vertices to account for the edges between them. The other disadvantage of such a representation was that for directed graphs, navigating the edges in the opposite direction was not readily possible. A complete database scan was required to traverse the edges in the inverse direction. In a graph database, such inversed path queries are far too common to adopt this approach to storing edges.

Research was also made to reduce the memory footprint by trying to compress the graph data. Though graph data is generally not homogenous, several characteristics like the degree, density follow a standard power law distribution [12] that can be exploited while compressing the data. Boldi ang Vigna [13] developed several compression techniques on web graphs including compression by gap encoding, interval representation and reference compressions. These techniques were especially effective on web graphs because most web graphs share a common vertex and exhibit community behaviors, both of which are useful characteristics while trying to compress data. This way, the information regarding a specific node or a community can be encoded in a smaller amount of data and this data can be used to represent the.

While such compression techniques helped in reducing the total size of the graph data, the issue of data locality was still present. Especially for navigation-intensive applications, it was desired that the graph vertex along with all its neighbors be represented together such that it can be retrieved in as many less disk reads as possible. For graph applications that used a breadth first navigation technique, Al-Furaih and Ranka [14] suggested storing the

graph by labeling the vertices in a breadth-first format. However, this technique would only work if the traversal always started from the origin vertex and performance decreased with the distance of the start vertex from the origin. Since graphs can be efficiently expressed as matrices, several matrix algorithms including bandwidth minimization techniques [15] were used to order graph objects for storage. GSpan [16] utilized a discovery based depth-first technique for graph storage. Both of these techniques do not fare well for updates, in which case the entire structure has to be re-written.

Most of these storage representations were dependent on the target application. While graphs themselves are generalized structures that could be adapted for several disciplines, the storage and retrieval methods need varied optimizations depending on the usage. STINGER [17] is a generalized graph representation format which was aimed to be universally applicable for several disciplines. The data model was developed to be portable between multiple languages and frameworks without compromising on performance and scalability. The storage structure comprised of an in-memory component and a disk-resident component with the in-memory component acting as an index for the graph structure. However, this model did not provide much optimizations for type-intensive graphs whose queries depended heavily on type-based constraints.

The graph model utilized also directly impacts the choice of graph representation. GraphGrep [18] assumes only vertices to have labels. The storage in GraphGrep consists of two tables, the label paths and a fingerprint table. The label paths stores the set of vertices belonging to specified types that have a path between them. The fingerprint table serves as an index for the entire graph, storing the number of paths that exist for a given label sequence.

2.2 RDF Databases

The development of the World Wide Web and semantic engineering also resulted in the large scale development of RDF data. RDF, short form for Resource Description Framework is a standard format for data interchange on the web. The data model consists of a set of

triples (subject, object, predicate) that tie objects with their references. The structure of RDF data forms a directed, labeled graph where the vertices represented resources and the edges represented the links between the resources. Using this simplified data model, RDF evolved as a primary model for storing unstructured data due to its flexible architecture and ease in merging and adapting to different schemas

RDF data storage and retrieval is a very active research field complete with specialized storage formats and dedicated access methods. SPARQL [19] has been adopted as the de-facto standard for querying RDF databases. The structure of SPARQL allows a direct mapping from SQL queries and hence has facilitated relational databases to be used as storage mediums for RDF data. Several storage systems were created that were optimized for storing and retrieving triple data structures. Some of the systems [20], [21] were built upon traditional relational stores that were optimized for RDF, while some others like [22], [23] were built from scratch especially for RDF data. RDF was marked by the huge volume of data it generated and hence scalability was an important part of RDF systems. Many systems like [24], [25] supported distributed architectures in which both the data storage as well as processing was distributed over a cluster.

It might seem natural to adopt the techniques on RDF systems and apply them into graph databases, given that RDF data is also graph-like. But there are quite some differences between RDF and graph data that prevent us from doing so. RDF can be considered as a simplified form of graph data that links up different resources. There is a regular pattern to the structure and to the queries. An RDF data does not support properties for its edges the way regular graph does. Hence, if we were to convert a general purpose graph into RDF, the attributes of objects are masked into separate relationships between the object and the attribute value. This prevents us from making deep, meaningful queries involving attributes, as is the usual case in a typical graph ecosystem. Also most indexing techniques for RDF data relies on the uniformity of the structure and hence do not apply well towards general purpose graphs. Hence, it is not possible to directly use RDF technologies for general purpose graphs.

RDF databases also distribute their data based on "predicate" values. However, such a distribution is not possible for general purpose graphs due to the imbalance in the type and number of predicates and the kinds of queries that would be made in a general purpose graph. The relationship structure in general purpose graphs also do not permit distribution based on predicates.

2.3 NoSQL Databases

The development of NoSQL databases helped address schema-less architectures and thereby created alternative methods to store graph data. Column based databases were and still are the primary storage back ends for high scale graph data including RDFs. Especially the BigTable databases like Cassandra [26], HBase [27] offered massive scalability and distributed processing support. Column databases differ from relational databases in that the data is stored in a columnar manner. This resulted in reduced seek time for queries since the locality of reference allows data to be found using minimal number of disk scans. Nevertheless, they were ill suited for traversal-intensive queries due to their logical representation.

Object oriented databases like Db4O [28] offered a storage representation as close as possible to the logical layout. For small to medium sized graphs whose entire data could be loaded into memory, these databases provided the best solution to store graphs. The logical representation was exactly the same as the physical storage representation and eliminated the need for any conversions in data for storage and performance reasons. However, for large data, the serialization and deserialization required multiple run time object packing and unpacking, and hence was not efficient even with advanced object mapping techniques.

Several other NoSQL systems including key-value stores like the BerkeleyDB [29] and document stores like MongoDB [30] were experimented as graph stores with limited benefit. The key factor affecting performance was the un-optimized data storage structure employed, which prevented graph-intensive queries from achieving better performance. Indexes were heavily relied upon which added to additional storage space and added additional overheads, especially during data inserts.

2.4 Graph Databases

Neo4J [31] was one of the leaders in developing a true graph database engine optimized to support storage and retrieval of graph data. Branded as a "native graph database", Neo4J developed a data model that stored each vertex, the data associated with the vertex and the edges associated with a vertex as a series of pointer connections. The model supported a full property graph with multiple edge types and multiple edges between its vertices. The database was divided into three main regions, one to store the vertices, one for the relationships and one for the properties. It attempted to provide a continuity between the logical and physical representations of the graph by combining the details of the vertex and the edge as a series of pointer directions.

The concept was to model the details of every graph object in an unbroken chain, in which each link is formed by a specific detail about the object. Each vertex record pointed to a property record and a relationship record if present. The relationship record in turn points to the previous and the next relationships of both the vertices involved in the relationship. Thus, to navigate a path in the graph in Neo4J, one would only have to narrow down the starting vertex. From then on, it would be a simple matter of following the pointer connections, alternating between a relationship record and a vertex record. Each record is of a fixed length and directly pointed to the physical location in the disk of the connecting record, rendering a constant time access O(1) to reach any record. Thus, the database adopted a completely index-free adjacency structure to store the graph, rendering even inverse queries straightforward. Once a specific relationship record is reached, the user can navigate either towards the head node or the tail node with no reduction in performance. A similar concept is adopted for properties too, where one property would lead to the next property of the graph object. Properties could be indexed by third party indexing systems like Lucene for improved performance.

The primary issue in this design was that it was not possible to obtain all the details of a given vertex in one consolidated place. Each additional edge or property required one

pointer reference and hence the time taken to collect all the information about any vertex in a graph with max degree Δ was $O(\Delta)$. This poses serious performance concerns in real world graphs with larger degrees. This was because, it was impossible to even know if a given vertex contained an edge of a certain type without going through all the relationship records that are associated with the vertex. Also, Neo4J heavily relied on cache performance to efficiently handle large scale data. Three levels of caching were utilized to limit disk access. However, due to edge-localization issues, it cannot be guaranteed that all the related information about a vertex can be obtained and stored in the cache beforehand. Hence, this posed a serious limitation on the DRAM space required for optimal performance.

Though Neo4J supported a distributed processing environment, the parallel processing was achieved through data replication instead of data partitioning in the true distributed sense. Simultaneous query processing was provided using multiple high availability clusters where the same data was replicated in multiple processors for simultaneous access. While this helped serving multiple user requests, it was not primarily aimed at processing a single query in a distributed fashion.

Dex [32] is another high performance graph database that uses a series of bitmaps to store graph data. The idea is to partition the entire graph into different segments such that each segment can be loaded into memory based on need. The storage model for DEX can be split up into three main sections, one for the graph objects, one for the relationships and the other for the attributes. The graph object section assigns unique identifiers to vertices and edges and uses a bitmap structure to group the objects by their types. The relationship segment utilizes two bitmap structures, one to store all the edges that have a particular vertex as their head vertex and the other one to group vertices at their tail. The attribute bitmap stores the graph objects that share the same value for the same attribute.

This way, the entire graph need not be in memory at all times. Depending on the query, parts of the graph can be loaded and accessed from memory. This method also voids the use of indices since the bitmap structures themselves serve as efficient indices for the graph. By grouping all the vertices that are the head/tail vertices for any given vertex together, the

system greatly improves path queries in both directions. In order to follow a path from the vertex, all we have to do is to reach the bitmap record pointed by the relationship bitmap corresponding to a given vertex. So, if we need to continue along the head node of a given edge, we examine the head bitmap and look up the vertex which is at the head node for the given edge. Now, to go to the next vertex in the path, we have the entire neighborhood of the head vertex to choose from. A similar approach can be obtained for the other direction too making the graph traversable in all directions. The other advantage of DEX was that joins could now be replaced by simpler set operations due to the use of bitmaps. Complex filtering operations could be done by set intersections between bitmaps, which are more basic operations.

Examining the data structure of DEX, we find that in order to navigate the entire graph, we need to move back and forth between the different bitmap structures. So, though the aim of segmenting the graph into smaller units is to create better memory occupancy, due to the need to alternate between the different graph segments, there is an additional time required for memory loading and unloading. Therefore, while DEX is extremely efficient in shallow search queries where the neighborhood of the query examined is small, for deep queries involving longer paths, the extensive memory allocations and de-allocations cause additional overheads. There is also the issue with the bitmap structures. As the size of the graph increases, it would be impossible to use an absolute bitmap reference for each graph object. Hence, it would be necessary to pack the bitmaps in a compact representation format. This conversion also creates additional overheads. Insertions and deletions also become cumbersome as it involves unpacking and repacking all of the bitmap objects involved.

OrientDB [33] is a hybrid NoSQL database that serves as a multipurpose document and graph database. It uses a JSON based structure to store information about the vertices and edges. The concept here is to treat the entire graph as a document, with the edges and vertices forming the individual items. There are two basic JSON structures, one to the store the vertices and the other to store the edges. The vertices document lists all the vertices in the graph along with their properties and connecting edges, both incoming as well as

outgoing separately as fields for the document. The edges referenced here are contained in another document housing the edges, where each edge lists down its head and tail node along with its properties.

So, in order to navigate the graph, the user would have to find the starting vertex, follow the edge to the edge document and find the vertex forming the other end of the edge and so on. One advantage of OrientDB is that unlike Neo4J and DEX, the properties of the graph are stored along with the graph objects to which they belong. In that sense, it stays true to the graph structure in which all the details about the graph object, be it the vertex or the edge, are stored together. Since it stores both the incoming as well as outgoing edges, it supports both forward as well as reverse traversals. However, there is a lot of back and forth that is required between the vertex and edge documents to navigate even a single path. Separate indexes are required to store the vertex and edge types and there is no way to directly find if the vertex has an edge of a certain type. Hence, additional overheads in terms of maintaining indexes becomes essential. So, a high performance degradation can be observed in the case of deep queries and also for graph related algorithmic operations like the shortest path and clustering.

2.5 Graph Indexing Techniques

Several innovative indexing models have been proposed for managing large graphs. Graphs are unique in that they are as much about the data contained as they are about the structure of relationships and without the structure, the data has no meaning. Relational databases have been around for a very long time and have developed efficient indexing methods to retrieve data. But the concept of relational indices are built around the idea that all the information associated with a given object is contained within the same table. Structural queries are usually performed through joins. Since the concept of a graph database was to eliminate the need for joins, much more specialized indexing methodologies are needed.

Structural queries come in many forms and most algorithms concerning graph structures are intractable. A good example is the subgraph isomorphism problem, which is NP

complete. To address this, a variety of structural indices have been proposed. The idea is to capture some information about the structure of the graph during graph creation time and later use this as a filtering criteria during query evaluation.

The most common type of graph structure indices are the path based indices. Some path information is extracted from the graph, which could be then used to solve frequent queries like the shortest-path and structure similarity. The fingerprint table used in [18] is an example of path-based index. Storing the labeled walk of the graph is yet another indexing technique as used in GRACE [34]. While these two techniques can efficiently index a large collection of small graphs, several indexing techniques were also developed to address a single large graph. Jin et al [35] used an edge-label based indexing technique to determine if there existed a path between two vertices such that the edge labels are totally contained within a set A.

The other way of indexing a graph structure is to extract substructures from the graph. Common data mining techniques such as frequent subgraph mining are used and combined together in a global indexing structure which are then used to filter candidates during query evaluation. gIndex [36] is a graph substructure indexing technique used for maintaining frequently occurring substructures in a database of labeled, undirected graphs. The concept of a graph closure was used in [37] to group vertices and attributes of a graph as a set of the vertices and attributes involved whereas [38] indexed frequently occurring tree structures. Spectral indexing methods were also employed for multi-dimensional and hierarchical graph databases [39], [40] by representing graph structures as vectors in a hypothetical space. GString [41] builds a structural index considering the semantics of the graph. Mainly aimed at graphs in the context of organic chemistry, it uses semantically meaningful structures like the Line, Star and Cycle to index the graph.

2.6 Distributed Graph Databases

As the scale of the data concerned grows, traditional single processor systems become obsolete, and it is imperative to adopt a distributed approach to data handling and process-

ing. Several attempts have been made to develop a completely distributed solution to graph data processing. NoSQL databases like Titan [42], OrientDB [33], Infinitegraph [43] have provided distributed capability for graph processing. Titan uses existing BigTable structures like Cassandra [26], BerkeleyDB [29], HBase [27] that are already equipped for distributed processing, as its storage backend. Thus, while Titan provides graph specific optimizations for querying, it is not a truly graph-oriented database and relies on off the shelf utilities for scalability. Infinitegraph builds upon its own distributed database engine ObjectivityDB to provide distributed support.

SPIDER [44] is a distributed system for evaluation of large scale RDF data. It uses the MapReduce framework of Hadoop [27] to distribute the data and well as the processing over a distributed environment. The RDF data is partitioned based on the URLs to preserve graph locality and reduce communication costs. Each graph partition holds a subgraph of the entire graph and queries are performed only against the subgraph contained. A further refinement step combines the results of all the partitions to produce the final result.

Gbase [45] also used Hadoop Mapreduce to distribute the storage and processing of large scale graph data. However, it attempted to use a unique storage representation by adopting "block compression" to store homogeneous sections of graphs together. It also used a special block placement strategy such that disk reads are optimized for the most common graph queries.

While these are persistence-based distributed solutions, other memory-based distributed solutions have also been developed. Trinity [46] is a distributed graph engine based on a memory cloud. It combines a shared memory cloud by combining the DRAM memories of several machines and utilizes a global index along with optimized memory management and network communication techniques to efficiently support online query processing and offline analytics on large graphs.

2.7 Distributed Graph Processing

While the above approaches aimed at developing the storage side of graph data, there are yet others that seek to improve query processing on graph-like data. These are techniques that shifted their focus from the data structure used as storage back end and instead seek to optimize querying methodologies for graph related queries, irrespective of the underlying storage format. Several techniques were used including dedicated distributed architectures like Trinity [46]. However, Hadoop and MapReduce opened up big graph processing to commodity architectures that were scalable and efficient. Several graph processing systems used the MapReduce strategy to define several Map and Reduce functions on the graph data to process and query the graph.

Surfer [47] and GBASE [45] are examples of systems that extend on MapReduce to process graphs more efficiently. Surfer used a propagation-based implementation model that shows good promise in edge-oriented graph tasks, but failed to perform well for vertex-oriented tasks. GBASE adopts a matrix-based processing of graphs with heavy dependence on I/O access.It relies on a vertex-centric approach and utilizes the Bulk Synchronous Parallel model. GraphLab [48] is yet another popular distributed graph framework that was written in C++. It follows an asynchronous distributed shared-memory abstraction which makes use of a multi-threaded execution model.

However, MapReduce is an I/O heavy architecture that relies on disk persistence after a map or reduce action. Therefore, natural implementations of iterative tasks like graph path queries were less efficient and, batch mode equivalents like joins were used to extract matching paths [49] [50]. Since joins are expensive operations, efficient indexing mechanisms [51] were required to speedup path queries.

The other option to reduce iterative communication in a MapReduce environment was to rely on extensive driver side aggregations [52] [53] Each partition computed a partial result based on the localized data and these partial results were aggregated together in the driver node. While this method reduced the need for extensive joins and eliminated the need

for frequent I/O, the heavy driver-dependence reduced parallelism and throttled the driver due to the high number of intermediate results that need to be processed. [54] implements regular path queries by partitioning the graph but not decomposing the query.

Google developed Pregel [55] as a vertex-centric parallel graph processing platform. It decomposed the operations into a sequence of steps, each of which can be carried out by a vertex independent of each other. Communication between vertices happened at the end of each super step. This model prevented race conditions and reduced inter-processor communication by restricting data flow between processors to a specific time period. Horton [56] is a distributed interactive query engine for large graphs. It comes with its own graph query language which provides better expression for reachability queries and provides a query execution engine that allows query execution in parallel. Giraph [57] was developed based on Pregel [55] introduced by Google. It relies on a vertex-centric approach and utilizes the Bulk Synchronous Parallel model. GraphLab [48] is yet another popular distributed graph framework that was written in C++. It follows an asynchronous distributed shared-memory abstraction which makes use of a multi-threaded execution model.

However, all of these systems either use specialized architectures or depend on the MapReduce paradigm to process graphs. Spark [58] is yet another distributed processing paradigm that provides an alternate solution to the disk persistence problems of MapReduce.

Parallel programming libraries like PBGL [59], STAPL [60] provide generalized APIs for performing popular graph operations like clustering, shortest path, centrality analysis, in a parallel fashion. Graph partitioning is yet another field that has seem sudden interest due to the rise in distributed computing. Graph Partitioning algorithms like the METIS [61] helped partition the graph data according to user defined functions to enable distributed processing approach. Several strategies like Vertex Cut and Edge cut partitioning are used in state of the art distributed graph processing systems to segment the input graph into several segments to reduce the number of edge crossovers.

PART 3

PROBLEM STATEMENT

The purpose of this research is to develop a graph database that scales well with graph size without compromising on performance. We propose to improve the performance of large graph databases by reducing the number of disk reads required for query processing. We also show that our unique solution can be extended to reduce communication overheads in a distributed graph setting and aim to develop a distributed model for graph querying.

As the size of the graph grows, a completely memory based system becomes impossible and disk based persistence and access becomes imperative. But, frequent visits to the storage device are major culprits in affecting performance, due to the cost of I/O operations involved. While significantly faster algorithms are being developed to process and analyze graph data, disk access time serves as the primary bottleneck in achieving good performance. Disk based access is prevalent and successfully used in all kinds of databases dealing in the scale of BigData. However, the techniques used to reduce latency due to disk access in the other databases cannot be directly applied to graph data. In all the other systems, latency is reduced by following two main techniques data locality and indexing. In all the other databases other than graphs, the **data** involved are the primary units of concern. Be it a relational database or a document database or a key value database, a query is often specified by introducing a few constraints on the target **data**.

Now consider the storage principles for these other databases. In most databases except the graph database, all the required information about the data can be stored together. Segments of data that are related to each other this can mean data with similar values for common fields are also stored close by in the physical disk through indexing mechanisms like hash index [62], B-Trees [63] and R-Trees [64]. Advanced methods are also developed for indexing textual data [65]. So, when a disk read is performed, most of the concerned

data is obtained in a single disk scan. Subsequent references to related data are therefore directly obtained from the cache storage, thereby reducing latency.

Such indexing methods are not possible in the case of graph data, the reason for this being the unique structure of graphs and graph queries in which the relationship between the data carries more significance than the data itself. Traditional indexing mechanisms are highly suitable for indexing value-based characteristics of data. However, these methods are not suitable for indexing graph data as there is no way to incorporate the structure of the graph within the index. The graph indexing methods discussed earlier try to handle a specific subset of the structure designed for the end application in mind. However, it cannot be considered as a generalized method for indexing graph structures in all scenarios. Moreover, most structural indices are bulky and building such extensive structure based indices adds additional overheads in data management.

Graphs can also not benefit from the locality of storage because they are three dimensional structures, where in, there exists no planar embedding for most large graphs. There is no way a vertex can be stored in a one dimensional plane such that all of its neighbors are equidistant from it. Consider the property graph shown in Figure 1.1. *Vertex 2* has three neighbors *Vertices 1, 2 and 4* connected by *edges e1, e5 and e4* respectively. If we were to capture this exact model in our storage system, then the edges *e1, e5 and e4* all have to be stored equidistant from *Vertex 2*. But this is not possible, since while representing this graph in one-dimensional file storage, the best we can do is to store two edges equidistant from any given vertex. This problem is exemplified as the size and density of the graph grows. Hence, it is neither possible to store the elements of the graph such that all associated data be retrieved in a single disk span, nor is it feasible to build comprehensive indices for every possible structural configuration.

The ultimate benefit of using a graph structure is to exploit the graph-like properties, wherein accessing a vertex's neighbors should be as instantaneous as accessing the vertex of question. A completely in-memory representation of the graph can provide such fast access due to register access. But, for large graphs with disk-based access, it is not possible to store

the entire graph in memory. In disk based systems, the concepts of pagination and caching play a great role in determining the performance. As discussed earlier, the peculiar structure of graphs causes edge-localization issues and prevent us from storing all required data about a vertex close to the source. Therefore, part of data is going to be stored all over the disk. This means, when a vertex is retrieved from the disk, not all the information associated with it is also going to be retrieved in one go. Hence, multiple disk accesses become necessary, leading to increased turnaround time for the query.

One possible method to overcome this is to distribute the storage and processing across several machines such that the entire graph segment contained within a processing unit can be completely stored in memory. This way, disk reads are eliminated, thereby improving performance. However, current distributed solutions pay heavily in terms of inter-processor communication time mainly due to the data structure and the distributed model involved. For graph queries, this is an especially limiting feature because, a graph query is both structure as well as data dependent. So, while parallelizing the graph across the cluster, without collective information about the entire graph structure in a centralized location, multiple shuffles are required between the processing nodes to completely gather all associated data about a vertex/edge. These extra shuffles cause additional delays in processing queries due to the serialization/de-serialization and network access that is required to communicate with the neighbors to and fro about the graph structures.

Also, graph queries are exploratory in nature. This means that, the continuation of a certain path along the graph is determined at run time depending on the structure and data properties of the graph object. Since not all communication can lead to successful query completion, it is imperative to identify and prune non-viable paths early on in the querying process. This saves significant time for deep queries involving large datasets by preventing unnecessary communication overheads.

However, graphs are traditionally considered as indivisible structures and hence there is no high level structure that can efficiently capture the critical details of the graph in a compact form. Having such an index structure would provide an abstracted graph representation

that can be examined to determine path continuation ahead of time.

In this research, we propose a novel data storage model for graphs that partitions the graphs into different segments, with each segment representing a different characteristic of the graph. We make the most critical segment lightweight enough that it can act as a generic index for the graph structure. Our model is extensible and generalized to most graph situations and would also be applicable in a distributed processing environment. With the proposed data storage format as the basis, we aim to build a completely distributed graph database that can store as well as process data in a distributed fashion without compromising on performance by reducing latency due to disk access time and inter-processor communication.

PART 4

A SCALABLE STORAGE STRUCTURE FOR BIG GRAPH DATA

Graph databases offer an efficient way to store and access graph like data. Their index-free traversal patterns make them suitable options for performing a variety of graph related operations. However, the unique structure of graphs poses challenges in designing efficient storage and retrieval mechanisms. This difficulty is highlighted for large graphs containing vertices and edges in the scale of big data. Existing methods either compromise on speed or resort to distributed environments to achieve scalability in size. In this paper, we propose a unique solution for a graph database that decomposes the graph data into three dimensions and adopts different storage mechanisms for each dimension to effectively store and perform graph operations.

4.1 Introduction

Graphs derive their importance from the representation of connected data, where the relationships are represented in a visual format. A graph database will not be as effective if this basic structure is not captured in its storage format. As long as the graphs are stored completely in memory, advanced object mapping techniques help preserve the fundamental structure of the graph. This, along with the fact that in-memory access is significantly faster than disk-based access speeds up graph queries.

But it is unrealistic to use a memory-based storage mechanism for large graphs due to prohibitive main memory costs. So, as the size of the graph grows, it becomes necessary to resort to disk-based storage for scalability and persistence. Now, in addition to incurring performance degradation due to frequent disk-access, there is also an impact due to the unique structure of graphs that prevent us from capturing the entire structural information as is, in a one-dimensional storage space.

Graphs have always been considered as consisting of two logical units the topology and the data. The topology region defines the connections between the graph vertices and edges and the data region defines additional attributes to the graph objects. If we examine the most commonly used graph queries, they are as much about the topology as they are about the data. Graphs have traditionally been viewed as indivisible structures and hence are not readily distributed unlike other tabular storage formats.

The other problem in designing a data model comes in the form of speeding up graph queries. If we consider the relationship between entities in a graph as its topology, and the entities themselves along with their properties as the data, then most queries in a graph databased are as much about the topology as they are about the data. A typical graph query in a Social Network system could be of the form "*find me all friends of a person named john who went to the same school as john*". This is a representative of both a Pattern Matching and Reachability query. If you want to translate this query in graph terms, you would say, "*find me all vertices of type* ***person****, who are connected by an edge of type* ***friend*** *to a vertex of type* ***person*** *having the property* ***name*** *as* ***john*** *and who have the same value for property* ***school***". The graph components are highlighted in bold. This query can be divided into its data and structural components. The data component of this query are given by `{name:john,school:same}`, and the topology component is given by `{vertex_type:person, edge_type:friends}`. More succinctly, this query can be represented as `Person [name:john,school:x] -- Friend--> Person[school:x]`, where the data components are enclosed within '[' and ']' and the rest of the query forms the topology region. We consider the type of the edges and vertices to be integral components of the entities themselves and do not view them as properties and hence are parts of the structural component.

Another query can be "*give me all vertices of type Person that have at least five friends*". This is a typical structural query, which finds more use in biological and chemical networks, where searching for a specific structure in the graph is the most common query (e.g. find me all carbon atoms that are connected to two oxygen atoms). In order to perform these

queries, it is essential to have simultaneous access to both the data and topology regions of the graph. If the graph is small enough, the entire data could be stored in memory and the problem is solved. But with substantially large graphs, which is becoming the norm for most applications, this is not a viable solution. Resorting to disk-based access means multiple calls are made to the system I/O to repeatedly extract information, which slows down the response time.

Graph have traditionally been considered to be indivisible units. Some solutions split the topology region from the property and use a disk based access for the properties, while the topology region resides completely in memory. This works because the structural region forms the first line of defense in answering graph queries and can eliminate most I/O calls to the properties [66]. But, this system also breaks when the graph is too large that the structural region could not be completely fit in memory. STINGER [17] developed a graph representation that encapsulated the in degree and out degree of a vertex as a separate logical entity that can aid in the filtering process. But it was not suited for label based queries which are the norm in most practical databases. In our work, instead of splitting the graph into two, we go one step further and partition the graph into three parts by dividing the topology region into two further parts. We strip the structural parts of the graph from the topology, creating two layers - the topology structure and the topology data. Now essentially, the graph is split into three regions the topology structure (TS), the topology data (TD) and the properties region. The TS region is kept bare bones and minimum so that it can completely be fit in memory. The TD and the property segments are stored on-disk, the property region using a traditional RDBMS store and the TD region using a special data model that we developed. The TS region forms the first filtering layer and greatly limits further I/O calls to the TD and property regions. Our system caters to attribute graphs where both edges and vertices can have properties. Our storage model is geared towards speeding most graph queries. Since only a part of the graph is stored in memory, it also improves scalability for large graph sizes. We propose that our method helps improve processing speeds for large graphs without sacrificing scalability. The initial

experiments conducted against the Neo4J database [31], a popular graph database explained earlier, prove that our method yields superior results in a variety of queries.

4.2 Data Storage Format

The key to fast query performance for any disk-based database, setting aside querying techniques and improved algorithms, is to limit the number of I/O calls. In spite of improvements in disk storage and access, access to secondary storage devices serves as the primary bottlenecks in data-induced latency.

One way to solve this problem would be to implement efficient caching and pagination mechanisms such that when data is accessed, most of the data that would be accessed in subsequent calls are also retrieved and stored in the cache. This way, during future calls to related data, it would be obtained from the cache instead of an I/O call. For a graph database, this would mean storing a vertex's neighbors and their neighbors along with their properties in memory locations closer to the target vertex. But, the inherent three-dimensional nature of graphs prevent us from localizing a vertex with its neighbors.

That being said, the other way to improve performance is through extensive use of indexes. Indexes can be built and optimized for a variety of needs and extensive research has been made in this field that can be tapped into. Considering the difficulty in building comprehensive indices for the entire graph structure, they are generally optimized towards a specific query and would not work in a generalized database where the range of queries are large. Also, as the size of the graph grows, the memory limits pose a check on the amount of indexes that can be had on memory.

In our research, we wish to improve query performance without trying to rely on cache performance and without the extensive use of indexes. We wish to preserve the actual structure of the graph as close as possible in the data storage model. Accordingly, we divide the entire graph into three parts the Topological Structure (TS), the Topological Data (TD) and the properties region. We consider the TS to be the backbone skeleton of the graph and hence offer it the highest position in order of importance. The properties region come next

and finally the TD region is accessed once the query clears through the top two regions.

The three segments of the graph the TD, TS and property define the three key dimensions of graph data. The property segment contains all the attributes associated with a graph entity. The TD segment gives information about the neighbors of a vertex and hence defines the relationship between the graph entities. We introduce a third dimension - the TS -which describes the structure of the graph without giving details about the actual neighbors. The graph topology is stripped of any information about the actual objects defining the relationship and a third dimension - the structure, that serves as the skeleton and placeholder for the actual objects is created. In essence we have normalized the graph data akin to a relational database. The three segments are explained in detail below.

4.2.1 Topological Structure (TS)

The topological structure forms the first line of defense for our query. It extracts the structure of the graph on a higher level of abstraction without going into details about the actual data elements. The data in the TS part gives you the overall skeleton of the graph without any information or reference to the graph entities. Passing a query through the TS eliminates most next level fetches to the properties and TD regions, both of which involve I/O access.

When we look at a graph from a human standpoint, the things that capture our attention are the vertex type, the type and number of edge of different types connected to the vertex and the presence or absence of properties without actual concerns about the exact vertex object or edge object. This gives a very high level perspective of the graph. This structural information is encapsulated in the TS region of our system. To describe the structure information of Vertex 1 in the graph given in Figure 1.1, we would say “Vertex 1 has property “name”, has one outgoing edge of type “Works For” with property “years”, one outgoing edge of type “Works With” and one incoming edge of type “Knows”. This information is captured into the TS part of Vertex 1. Each TS entry is of the form shown in Figure 4.1

The first byte represents whether the vertex is deleted or active. The next bytes gives

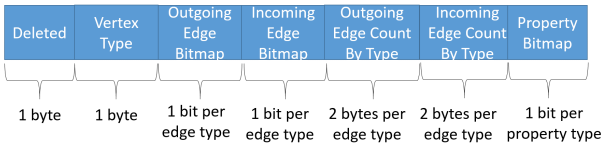


Figure (4.1) Topology Structure Region

information about the type of the vertex. This allows our system to represent up to 2^8 different vertex types. The next two fields are bitmaps that represent the presence or absence of an edge type by a 1 or 0 respectively. The next field gives a count of the number of outgoing edges of each type connected to the given vertex. Each count is represented using 2 bytes. The next field gives a count of the number of incoming edges of each type connected to the vertex. This again uses 2 bytes per count. So, the total number of edges that are supported by our system for each type are 2^{16} each for incoming and outgoing for every edge type. The counts are placed in the same order as the edge types.

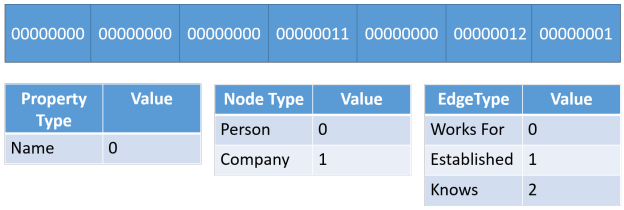


Figure (4.2) Topology Structure Example

The size of the last three fields of the TS vary with the number of different types of edges the system can support. We believe, these parameters are design time decisions and have to be set before creation of the database. However, since the size of our TS region is quite small, it wouldnt take long to update the fields for all the vertices if required. That way, additional edge types can easily be incorporated into our system.

For the number of edge types fixed at 8, the structure record for vertex 2 is given in Figure 7.1. Note that the incoming and outgoing edge counts are expressed as words (4 bits)

for brevity. The node and edge types for Figure 1.1 are given in the tables by the side. The same order of edge types is maintained throughout the database for proper correlation with the topology data.

The TS data resides completely in memory. Consider a graph with n vertices, n_e different edge types and n_p different property types for the vertices. The total size for each record in the TS region in bytes is given by:

$$\begin{aligned} size =& 1 + 1 + \frac{n_e}{8} + \frac{n_e}{8} + 2n_e + 2n_e \\ =& 2 + \frac{17}{4} n_e \end{aligned}$$

So, for a system with $n_e = 8$, the size of each record in the TS region is 36 bytes. So, for 1 million vertices, this is only going to take 36MB of data to store the TS region of all the vertices, which is quite a small size. The other interesting aspect of our TS system is that the total size only depends on the number of vertices and the number of edge types, and is agnostic of the number of edges. In any typical graph, the number of edges is significantly higher than the number of vertices and in this way, our system performance is independent of the density of the graph.

4.2.2 Topology Data

The TS region is incomplete without knowledge about the actual vertices and edges of the graph. This is provided by the TD region. The TD region works in conjunction with the TS region to decipher the complete topology of the graph. Each record in the TD section describes the neighborhood of a vertex and contains a pointer to the physical position of the neighboring vertices.

The order of the neighboring vertices follow the same sequence as described in the TS file. That is all the outgoing vertices are stored first, again following the sequence given in the TS record, followed by all the incoming vertices, again in the same order.

In order to enable fast lookups in the database file, we make each record size a constant in the TD file. The position of each record in the TD file corresponds to the id of the vertex. So, the record for vertex with id=5, for example, is stored at position 5 in the TD file. This way, navigating to any vertex record is made easier as it corresponds to the id. Now the actual position of a vertex can be calculated by multiplying the position with the record size. Hence record position calculation takes $O(1)$ time instead of the $O(logn)$ time that would be required for performing a search.

But, maintaining a fixed record size for each vertex is not straightforward as the number of neighbors for any given vertex can vary hugely with each other. In real graphs, we have seen this variation to be from 0 to thousands of neighbors. In order to fix this, we set a limit on the maximum number of neighbors each TD file can handle. If, for a vertex this limit is exceeded, then we create another TD file to contain the spillover neighbors.

The newly created TD file also follows the same structure as the initial TD file. The spillover neighbors of vertex with say, id=x, will still be stored at position x of the new TD file. So access of neighbors does not differ from the original file to the newly created files as the record to be obtained is located at the same position.

Now, if another vertex, with id=y, wants to store additional neighbors, it can store at position y of the TD file that was already created to store the extra vertices of vertex with id=x. Therefore, if the max degree of the graph is, and if the threshold for the number of neighbors in each TD file is, then the total number of TD files created would be.

In order to properly access the neighbors in all the overflow files, the correlation with the TS file is maintained for each record across its entries in all the TD files. That is, the records for the vertex across all the TD files are treated as one large record. This way, the order of vertices is matched with that of the TS file. Therefore, in case of an insert of a neighbor later on, the neighbor vertex is inserted at the position that corresponds to the edge sequence and the rest of the neighbors are pushed on to the next TD files.

We use 4 bytes to store each neighbor. Which means, right now our system can handle up to 2^{32} different vertices. The type and direction of the edge are determined from the

TS file as the records of the TD file directly correspond to the TS file. So, we do not have any detail about the neighbor vertex stored in the TD file. As mentioned above, in order to decode the topology of the graph, the TD and the TS files work in tandem.

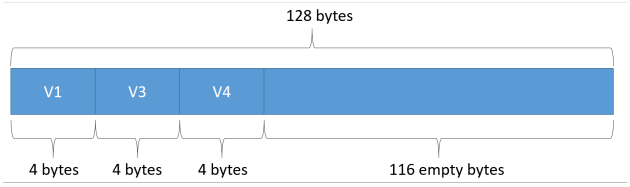


Figure (4.3) Topology Data

The topology structure record for *Vertex 2* of Figure 1.1 is shown in Figure 4.3. The structural information of *Vertex 2* is required in order to decode the structural region entry. The TS data for *Vertex 2* gives us the information that this vertex has no outgoing edges and has *3* incoming edges, of which *2* belong to edge type *0* and *1* belongs to edge type *1*. Hence in the given TD record, the first two entries belong to vertices connected to *Vertex 2* by an incoming edge of type *0* and the last entry indicates that the vertex is connected to *Vertex 2* by an incoming edge of type *1*. If the max number of neighbors in in each TD file is set to be 32, then the size of each TD record Δ, is 128 bytes. So, in the case of *Vertex 2*, since it has only 3 neighbors, the first 12 bytes are used to store the neighbor positions and the last 116 bytes are left empty. This space can be used up if additional neighbors are added to *Vertex 2* in future before spilling over to additional TD files.

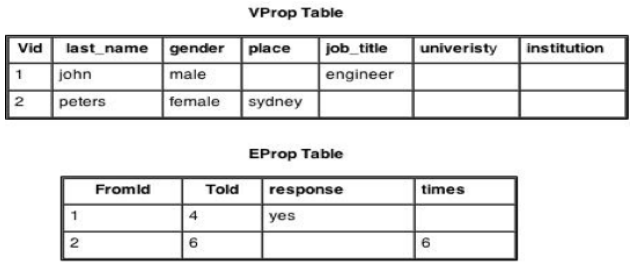
4.2.3 Properties Region

The property region forms the third segment of the graph. Both vertices as well as edges can have properties. The property information is stored in an RDBMS system with separate tables for the vertices and edges. RDBMS has been around for decades and lot of research has been put into making the system as efficient as possible. While we can argue that an RDBMS is not the most efficient structure for storing graph databases due to the

number of joins required to uncover relationships, it is optimized for storing tabular data.

Consider the properties of a graph entity, be it a vertex or an edge. These are inherent attributes of the entity and have no relationship between other entities in the graph. When we try to extract all the properties of an entity, which is a very common query in a graph database, we would want all the properties of an entity to be stored along with the entity itself. This is the ideal scenario of an RDBMS that follows a Row-Major order.

The other kind of query on properties could be to extract all the vertices that satisfy a given set of properties. This is also easily done in an RDBMS by making use of the efficient indexing and optimization techniques developed for RDBMS. Hence we justify our use of an RDBMS to store the properties of the vertices and edges of our graph.



VProp Table

Vid	last_name	gender	place	job_title	univeristy	institution
1	john	male		engineer		
2	peters	female	sydney			

EProp Table

FromId	ToId	response	times
1	4	yes	
2	6		6

Figure (4.4) Sample RDBMS to store properties

We have two separate tables to store the properties for vertices and edges. The VProp table contains the id of the vertex as its primary key, along with all the properties as individual columns. We adopted this approach as opposed to normalizing the properties across different tables to eliminate joins in the database. Since our queries to the RDBMS are only to extract the properties given the id and the id given the key-value pair of the properties, this approach works fine for us. The only time this system suffers is when a new property has to be added in which case the entire database has to be updated. Also, this way of storing does generate a sparse matrix structure and increases storage size. But at this point, we are not worried about storage and are more concerned about querying performance. We are also experimenting with adopting a normalized approach to manage

these problems.

A similar approach is adopted for the edge properties. The EProp table contains the columns fromId and toId, which together form the primary key of the table. They denote the tail and head vertex for each edge. This way, edges do not have separate ids to identify themselves. Instead they are identified by the two vertices involved in the edge. The properties for the edge are stored as individual columns. The structure of our RDBMS is shown in Figure 4.4.

4.3 Query Evaluation

The reason towards splitting the graph data into three parts is to speed up query evaluation for a range of graph queries. Let us consider the basic structure of a graph query. A typical graph query, be it a path query, reachability queries, k-hop queries can be abstracted to be of the form given in Figure 1.2. Each entry contains the type of the graph entity (vertex/edge), the edge direction and the optional properties that the entity has to satisfy.

Splitting the graph into its properties and topology as done by Sakr et al [66] lets us first check the in-memory component, which is the topology of the graph. If this test passes, then the property region is accessed to further extract relevant elements. This approach works well if the entire topology data can be loaded into memory as was the case of by Sakr et al [66]. But when this is not possible, which happens to be the case for large graphs, the secondary storage unit has to be accessed to perform the query, which causes serious performance drops due to the multiple I/O calls. Our method of splitting the topology data into the TD and the TS parts helps eliminate this problem. The TS data, which captures the high level abstraction of the structure of neighbors of a vertex is always on memory and hence can be checked first to ensure if a certain path down the graph has to traversed at all. This sort of creates an exploratory querying in which only those paths that have a high probability of reaching the result are pursued while the others are dropped. For example, suppose we want to query all the neighbors of a vertex connected by a given edge,

knowing that the vertex has no edge of the given type stops the query right there. This saves significant time in terms of unwanted calls to the storage unit. In our experiments, we found that this resulted in significant gains in query performance.

At the start of the execution, the Topology Structure region is loaded into memory, using the following data structure for each record:

```
{
  byte deleted;
  byte vertexType;
  byte incomingEdgeBitmap;
  byte outgoingEdgeBitmap;
  short[] outgoingEdgeCount;
  short[] incomingEdgeCount;
  byte propertiesBitmap
}
```

We used a bitmap to store the presence or absence of an edge type to obtain improved performance in two fronts. First, it becomes a constant time access to check for the presence of edges for each edge type. Second, this allows only those edge types which are connected to the vertex to be loaded in memory in a packed array format. This saves a lot of space on memory because not all vertices have all types of edges connected to them. So, an edge count is loaded onto memory for a vertex only if it has at least one edge of that type. The edge count of those edge types that are absent for a vertex are not loaded into memory. This way, even though the size of each Topology Structure record is 37 bytes, it is the maximum value and in reality takes up much less space when stored in memory.

When a graph query is encountered, the Topology Structure region is first queried to retrieve a set of vertices that satisfy the vertex type and the edge types. The Topology Structure region stores the number of edges of each type that the vertex is connected to. An additional index stores the vertex numbers corresponding to each Vertex type. The execution plan for all graph queries is as follows:

1. Query the Topology Structure region to get the list of vertices that satisfy the first

Vertex, Edge Type and Properties in the query

2. If the query contains properties, query the RDBMS on the set of vertices obtained from Step 1 and obtain a subset of the input vertices that satisfy the properties

3. Return the neighbours from the Topology Data section for those vertices returned by Step 2.

4. Return to Step 1 for the next Vertex-Edge branch in the query

As we see, Steps 2 and 3, which access the secondary storage unit, only work with a subset of vertices obtained from the previous step, as opposed to the entire graph. Also, due to our implementation of random access methods in the Topology Data section, retrieval of edges in Step 3 is also accelerated, resulting in a shorter turnaround time. This query evaluation method remains the same for different kinds of queries like the path queries, k-hop queries and structural queries. Accesses to the secondary storage unit to retrieve the TD and property sections were limited to the queries listed in Figure 4.4 and Figure 4.6.

Query	Description
Select id from vprop where id in <id_list> and <prop1=value 1> and <prop 2=value 2,... and <prop n=value n>	Returns a subset of vertex ids from a given set that satisfy the given optional set of properties
Select prop1, prop2,... prop n from vprop where id=<vertex_id>	Returns the values of the given properties of a given vertex

Figure (4.5) Queries to the RDBMS

4.4 Experimental Results

We tested our method on an i3 processor with 6GB RAM and 256GB Secondary Storage. The entire experiment was programmed in Java 7 and MySQL 5.5 was used as the RDBMS.

Query	Description
getNeighbors (int id, Direction direction, String ... edgeType)	Returns the neighbors of a given vertex that are connected by a given edge type and in the given direction

Figure (4.6) Queries to the TD region

We used the GPlus dataset from the SNAP [12] website. This dataset consists of 107614 vertices and 13673453 edges. The vertices and edges did not have specific types. So we randomly generated vertex and edge types using a maximum of 8 vertex types and 8 edge types.

4.4.1 Memory Utilization

In order to measure the relative performance of our system with established methods, we tested our system against the popular Neo4j [31] database. We did not compare our system against G-SPARQL [66] because G-SPARQL used an in-memory execution technique for the entire topology region, whereas ours only used a portion of the graph on memory while most part of the graph resided in the hard disk. However, we easily prove that our system is more scalable than G-SPARQL by calculating the memory requirements. G-SPARQL used a pointer based representation to store the topology information. We were not able to get the exact implementation details. But assuming the lowest values possible, at least 4 bytes are required to store the actual vertex. Edges can be stored as pointers to the node objects using 5 bytes each (4 bytes for the vertex, 5 bytes each for the previous and next pointers. 5 bytes are needed at the minimum because 4 bytes will be required for the pointer information and 1 byte will be required for storing the edge connection type). This is again a stripped down version to calculate the minimum costs. Since we are concerned only with the online query performance, we did not compare our system with offline analytical engines like Pregel [67], Boost Graph Library [68] and GraphLab [69]

The size of the TS entry in our case is 36 bytes at its maximum. It might seem that our implementation is less memory efficient than G-SPARQL. But there are two factors that are to be considered. First, even though our TS entry is 36 bytes, it is the max value and most of the entries are much lesser than this. Second, our implementation is independent of the number of edges, whereas the number of pointers required in G-SPARQL is equal to the number of edges. So, as long as the edge to node ratio is less than 3, our method has a higher memory consumption than G-SPARQL. But once that threshold is crossed, our system significantly outperforms G-SPARQL in terms of memory usage.

Most real world graphs are dense in the sense that there are significantly more number of edges than there are vertices. Especially, graphs in the social network space have edge to node ratios in the scale of 10-100 [12]. Consider for example the dataset used in this experiment. The GPlus dataset has 107614 vertices and 13673453 edges. Calculating the memory requirement for the in-memory portion of both the graphs, our system takes (107614*37) 3981718 3.98 MB. G-SPARQL on the other hand takes (107614*4) + (13673453*10) = 137164986 = 137.2 MB. This stark difference in performance continues as the density of the graph increases. In general, our scalability only depends on the number of vertices whereas G-SPARQL depends on both the number of vertices as well as the edges. The difference in memory utilization is given in Table 4.1.

4.4.2 Query Performance

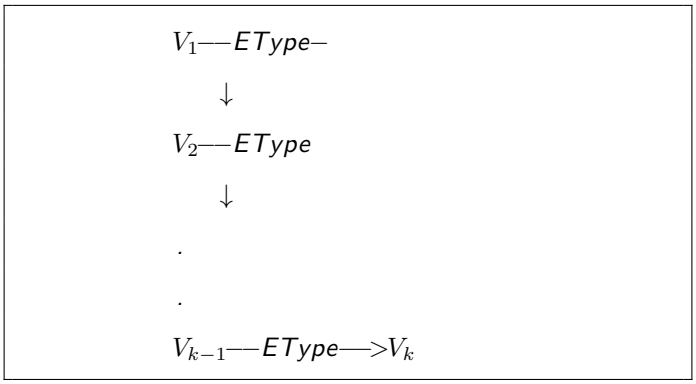
We compared our system performance using 4 different types of queries.

1. Q1 is a typical sub graph pattern matching query of the form

$$
\begin{array}{l}
V_1\,Type[V_1\,Props]--E_1\,Type[E_1\,Props] \\
\qquad\qquad\qquad\qquad\downarrow \\
V_2\,Type[V_2\,Props]--E_2\,Type[E_2\,Props] \\
\qquad\qquad\qquad\qquad\downarrow \\
. \\
. \\
V_n\,Type[V_n\,Props]--E_n\,Type[E_n\,Props]
\end{array}
$$

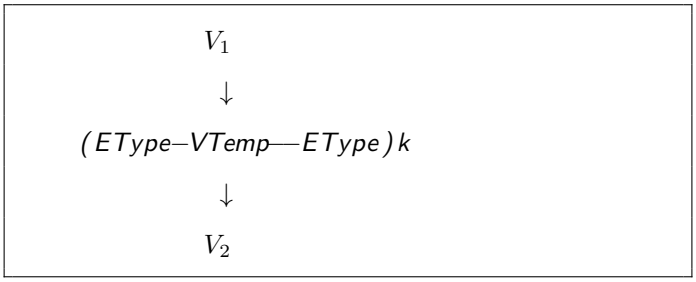
Here, each vertex-edge branch has a vertex and edge type with optional properties. We varied the path length from 2-6. We were able to generate many different kinds of paths with this template by varying the ordering of the properties and connection type. We also generated a few queries to represent cycles, triangles, rectangles and other common shapes. An example application of a pattern matching query in a recommendation system can be *"Find all people who know* ***John*** *and also watched the movie* ***Titanic*** *and listen to* ***Heavy Metal****"*. Such queries can easily be simulated using the path structure given above.

2. Q2 is a k-hop query that gets the k-hop neighborhood of a given vertex. It is of the form



We varied the path length from 2-6. K-hop queries are common in social networks to find the friends of friends of a person.

3. Q3 is a reachability query which tests for a path between two vertices. A reachability query can be denoted by



where V1 is the starting vertex and V2 the end vertex. An example of the reachability query is "is there a path between vertices v1 and v2 of length k such that they are

only connected by edges of type EType".

4. Q4 is a structural query which tries to find vertices that belong to a specific pattern. An example is of the form

$$VType{-}{-}\{E_1\,Type{=}n_1\,, E_2\,Type{=}n_2\,\ldots\,E_k\,Type{=}n_k\}$$

Structural queries are crucial to mine star patterns in a graph with restrictions on the number and type of vertices connected to the center. These are key queries to answer the structural properties of the graph like the maximum degree, identifying leaf vertices, etc.,. These queries are especially handy in bio-chemical networks to identify molecular patterns.

These queries were instantiated 20 times with random values. The values were recorded in a log file and the each database was queried with the same values to maintain uniformity while testing. The cache was also cleared for each of the systems using random values to remove bias while running the queries. Figure 4.7 shows the comparison of our system against Neo4J. Note that the time is represented using logarithmic scale.

Table (4.1) Memory Utilization

Our System	**SPARQL**
3.98 MB	137.2 MB

As expected our system outperforms all the other systems in all these different kinds of queries. Figure 4.8, which gives the average trips to the secondary storage device explains the reason for our improved performance. Our efficient use of the graph structure in query evaluations prevents most trips to the hard disk, thereby reducing as many I/O calls and hence the improved performance. Q4 shows the most gain in performance for our system because the entire query is answered from the main memory without ever having to access the storage device. However, our solution does not truly scale well for big data consisting of billions of graph objects. Our in-memory TS region poses a limit on the maximum number of

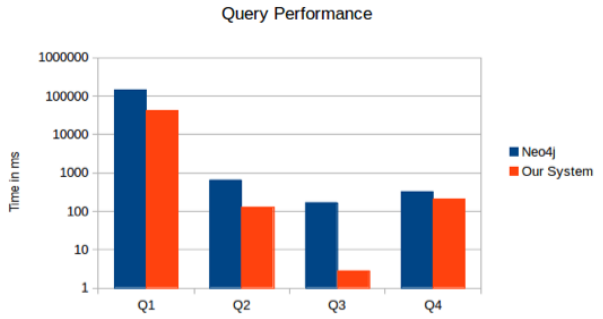


Figure (4.7) Query Performance comparison with Neo4J

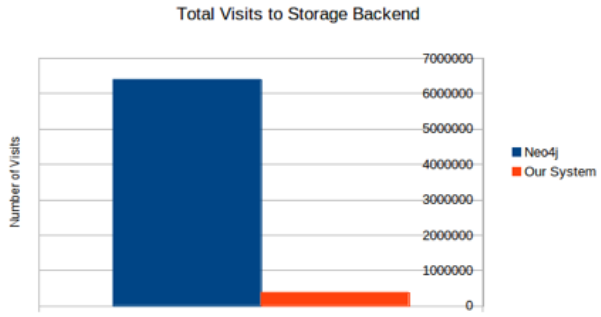


Figure (4.8) Total Number of visits to Storage Device

vertices that could be handled. Hence, we are working on advanced caching and replacement methods that lets us keep only a part of the TS region in memory while the rest can stay on disk.

4.5 Conclusion

In this section, we presented the design of a high performance graph database that efficiently stores and processes graph data in a larger scale. We are still implementing this design and are working on fine tuning the querying process for a large graph setting. We believe, segmenting the data into structural skeleton, structural data and property regions helps for three separate layers of abstraction, which work together in reducing the number

of I/O calls to the disk. By maintaining fixed length records for the structural data region, we convert all data reads to random access reads which further improves efficiency.

PART 5

SCALABLE CACHING FRAMEWORK

In our previous section, we described a graph storage structure that splits the graph into three logical components to improve performance of online pattern matching queries. One of the graph components was, however, maintained completely in memory to reduce I/O access. However, by keeping this structure memory-resident, our structure was able to support only mid sized graphs.

In this section, we eliminate the memory constraints and develop a scalable storage structure for large graphs. Our data model divides the entire graph into three logical layers, with the first layer forming the structural abstraction for the entire graph. Instead of maintaining this structure completely in memory, we use an adaptive caching process to maintain only those parts that are integral to the query in memory while the remaining are stored on the disk. The caching scheme ensures that our storage model is theoretically scalable for any graph size, while also ensuring that information most relevant to the query stays on memory. We compared our system with Neo4J and were able to maintain our performance improvements from [70] without compromising on scalability.

5.1 Caching Strategy

The reason behind splitting the graph into three layers is to better position them in memory and restrict access to the on-disk components. The Properties and the Topology Data region form the secondary bulk of data describing the graph and are hence stored completely on disk. The Topology Structure region on the other hand is designed to be as minimalistic as possible and serves as the first line of defense to answer queries. In the previous section, in order to completely eliminate disk access for the Topology Structure region, we stored this component completely in memory. But this posed a limit on the

maximum number of vertices the system can hold. Assuming the number of edge types to be 8 and the number of property types to be 8, each record of the Topology Structure region requires 37 bytes of data. So, to store the Topology Structure information for a million vertices, we would need 37MB of data and 37GB for a billion vertices, which is not an acceptable memory requirement. Therefore, in order to achieve true scalability, we adopted a hybrid disk-memory architecture to store the Topology Structure region. Only those parts of the Topology Structure region that have a higher probability to be used in the query are stored in memory and the rest remain on-disk to be obtained on an on-demand basis. Since graph exploration is memory-intensive, we allocated 35% of the available main memory to store the in-memory portion of the Topology Structure data, leaving enough free memory for the other graph operations. Since the Topology Structure is designed to prune paths that will not satisfy a given query, those vertices that have a higher probability to succeed, are preloaded onto memory. We used the degree of the vertex as the initial heuristic to accomplish this. We ordered the vertices in descending order of their degree and loaded the Topology Structure records of the vertices with higher degrees in memory until the memory limit of 35% is reached. We used a hash map structure to store the records in order to achieve O(1) lookup.

Now, when a query is issued, the caching algorithm updates the hash structure by flagging those preloaded vertices that do not conform to the query. When a vertex required for the query is not found in the cache structure, it is read from the disk and swapped with one of the vertices flagged for swapping. The decision to flag a vertex for a swap is as follows:

1. Priority 1 : All vertices whose vertex type does not form part of the query
2. Priority 2 : All vertices whose vertex type forms part of the query but whose neighbourhood does not conform to the query
3. Priority 3 : The least recently used vertices

Graph path queries typically contain constraints about the vertex and edge types. Since we look for exact matches and not approximate matches, if a vertex type is not mentioned

in a query, those vertices belonging to the said type can be safely eliminated from the search space. Similarly, those vertices whose structural definition does not match the query can also be marked as unnecessary.

As a last strategy, the vertices that have not yet been part of the result are marked for swapping. The idea here is, if a vertex did not satisfy the conditions of a query the first time, it is not going to satisfy the conditions during the further rounds. When a substructure query is issued to a graph, the result graph is grown from many viable seed vertices. Since, our execution model is sequential and we follow a Breadth First Search(BFS) approach for growing the subgraph, a second seed is not grown until the current one is explored. Hence, if a graph vertex is not requested in the initial few seed growths, the probability of it being used in later rounds diminishes with every seed vertex that it is not a part of. The Least Recently Used(LRU) heuristic is ideal for such situations, since it constantly replaces the least used vertices with ones that are more viable for the given query.

In order to prevent unnecessary scans, a vertex from the hashmap is marked for swap as and when it is deemed unnecessary in the course of the query. This cache replacement strategy is adaptive to the query and ensures that the vertices that are more likely to be part of the result are directly available in memory. The model follows an "*eventually complete*" paradigm where the index is optimized with every new seed branch of query computation. Each initial path computation for the query brings the Topology Structure closer to its stable form. The initial latency due to I/O reads to update the hashmap are balanced by the broader reduction in latency for the rest of the queries.

5.2 Query Execution Plan

Our query starts with the Topology Structure Region that is stored as a hash map within the main memory, with vertex id as the key and Topology Structure record as the value. The Topology Structure region is kept partially in memory and hence can be checked first to ensure if a certain path down the graph has to be traversed at all. This creates an exploratory querying structure, in which, only those paths that have a high probability of

reaching the result are pursued while the others are dropped. For example, suppose we want to query all the neighbors of a vertex connected by a given edge type, knowing that the vertex has no edge of the given type stops the query right there. This saves serious time in terms of unwanted calls to the storage unit. In our experiments, we found that this resulted in significant gains in query performance. When a graph query is encountered, the Topology Structure region is first queried to retrieve a set of vertices that satisfy the vertex type and the edge types. The Topology Structure region stores the number of edges of each type that the vertex is connected to. An additional index stores the vertex numbers corresponding to each Vertex type. The execution plan for all graph queries is as follows:

1. Query the Topology Structure region to get the list of vertices that satisfy the first Vertex, Edge Type and Properties in the query

2. If the query contains properties, query the RDBMS on the set of vertices obtained from Step 1 and obtain a subset of the input vertices that satisfy the properties

3. Return the neighbours from the Topology Data section for those vertices returned by Step 2.

4. Return to Step 1 for the next Vertex-Edge branch in the query

As we see, Steps 2 and 3, which access the secondary storage unit, only work with a subset of vertices obtained from the previous step, as opposed to the entire graph. Also, due to our implementation of random access methods in the Topology Data section, retrieval of edges in Step 3 is also accelerated, resulting in a shorter turnaround time. This query evaluation method remains the same for different kinds of queries like the path queries, k-hop queries and structural queries.

5.3 Experimental Results

We tested our method on an i3 processor with 6GB RAM and 256GB Secondary Storage. The entire experiment was programmed in Java 7 and MySQL 5.5 was used as the RDBMS.

We used the LiveJournal dataset from the SNAP [12] website. This dataset consists of 4848571 vertices and 68993773 edges. The vertices and edges did not have specific types. So we randomly generated vertex and edge types using a maximum of 8 vertex types and 8 edge types.

In order to measure the relative performance of our system with established methods, we tested our system against the popular Neo4j [31] database. We did not compare our system against G-SPARQL [66] because G-SPARQL uses an in-memory execution technique for the entire topology region, and hence is not scalable. Due to the hybrid memory storage of the Topology Structure region, our system is theoretically able to handle larger graph sizes. Since we are concerned only with the online query performance, we did not compare our system with offline analytical engines like Pregel [67], Boost Graph Library [68] and GraphLab [69]

5.3.1 Data Ingestion

Since our storage structures rely on a methodical ordering of the entire neighborhood of each vertex, data ingestion is faster for bulk loads rather than incremental updates. The data load time comparison for the LiveJournal dataset is given in Figure 5.1.

5.3.2 Performance Measurements

We compared our system performance using 3 different types of queries.

1. Q1 is a typical sub graph pattern matching query of the form

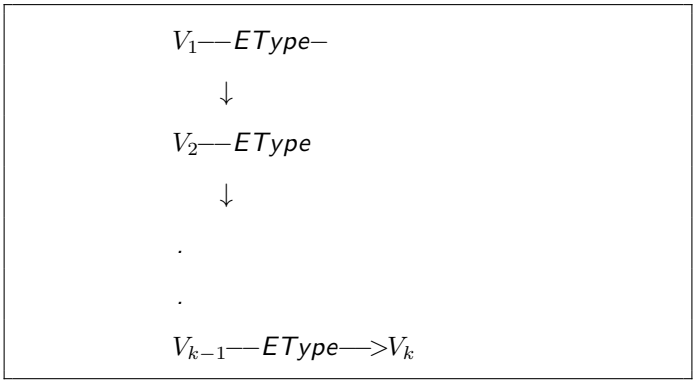
$$
\begin{array}{l}
V_1\,Type[V_1\,Props]--E_1\,Type[E_1\,Props] \\
\qquad\qquad\qquad\qquad\qquad\downarrow \\
V_2\,Type[V_2\,Props]--E_2\,Type[E_2\,Props] \\
\qquad\qquad\qquad\qquad\qquad\downarrow \\
. \\
. \\
V_n\,Type[V_n\,Props]--E_n\,Type[E_n\,Props]
\end{array}
$$

Here, each vertex-edge branch has a vertex and edge type with optional properties. We varied the path length from 2-6. We were able to generate many different kinds of paths with this template by varying the ordering of the properties and connection type. We also generated a few queries to represent cycles, triangles, rectangles and other common shapes. An example application of a pattern matching query in a recommendation system can be *"Find all people who know **John** and also watched the movie **Titanic** and listen to **Heavy Metal**"*. Such queries can easily be simulated using the path structure given above.

Table (5.1) Data Ingestion Time

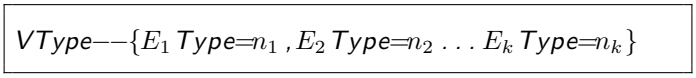
Neo4J	**Our System**
3795.6 ms	474.4 ms

2. Q2 is a k-hop query that gets the k-hop neighborhood of a given vertex. It is of the form



We varied the path length from 2-6. K-hop queries are common in social networks to find the friends of friends of a person.

3. Q3 is a structural query which tries to find vertices that belong to a specific pattern. An example is of the form



Structural queries are crucial to mine star patterns in a graph with restrictions on the number and type of vertices connected to the center. These are key queries to answer the structural properties of the graph like the maximum degree, identifying leaf vertices, etc.,. These queries are especially handy in bio-chemical networks to identify molecular patterns.

These queries were instantiated 20 times with random values. The values were recorded in a log file and the each database was queried with the same values to maintain uniformity while testing. The cache was also cleared for each of the systems using random values to remove bias while running the queries. Figure 5.1 shows the comparison of our system against Neo4J. Note that the time is represented using logarithmic scale.

As expected, our system outperforms Neo4J in all these different kinds of queries. Figure 5.2, which gives the average trips to the secondary storage device explains the reason for our improved performance. Efficient use of the graph structure in query evaluations using the Topology Structure, prevents most trips to the hard disk, thereby reducing as many I/O calls and hence the improved performance. Q3 shows the most gain in performance for our system because the query is answered entirely from the Topology Structure region without ever having to access the storage device.

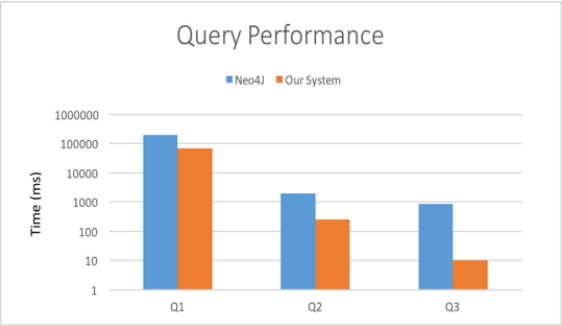


Figure (5.1) Query Performance

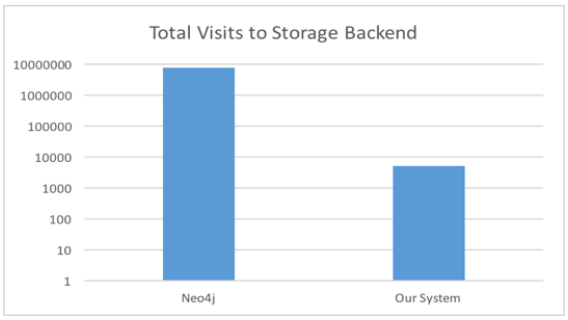


Figure (5.2) Average trips to secondary storage

5.4 Conclusion

In this section, we presented an optimized storage system that uses an adaptive caching technique to improve querying time for big graphs without compromising on scalability. We presented a three-pronged storage approach where the graph data is divided into three parts- the Topology Structure, the Topology Data and the Property. The Topology Structure helped improve query performance by limiting the number of trips made to the storage unit. The adaptive caching strategy helps utilize the benefits of in-memory access while at the same time maintaining scalability. Hence, our system is virtually scalable for graphs of any size without constraints on main memory. Our comparisons with Neo4J also show that our system does not lose performance while trying to address scalability concerns.

PART 6

DISTRIBUTED GRAPH PATH QUERIES USING SPARK

Graphs are increasingly being used as the data structure of choice to represent interactions between heterogeneous entities. Graph path querying is a primary operation in the network graph space, for both real time querying and inferential analysis. The rate and volume of interconnected data being generated warrants efficient distributed solutions to manage and query network graphs in a scalable fashion. Existing distributed solutions have proposed several optimization techniques, including intelligent joins and partial evaluations to process path queries. However, the former relies on comprehensive indices while the latter involves extensive driver-side processing to combine the partial results, neither of which is efficient for processing large graphs. In this section, we propose a novel distributed graph path query processing system using the Apache Spark framework.

6.1 Introduction

Most graph processing frameworks adopt a "triple-based" storage structure", where the graph is represented as individual tuples of the form $< subject, predicate, object >$. However, this is a highly compartmentalized representation of graph, where the collective graph information can only be obtained by combining individual tuples through a series of joins. Hence, extensive indexing is required to restrict the search space for graph traversal.

In this section, we describe a novel distributed graph path querying framework that works without the need to build extensive indices. To the best of our knowledge, this is the first Spark-based implementation of graph path queries.

6.1.1 Spark

Spark is a distributed data processing framework developed by the University of California, Berkeley. It was developed to address the shortcomings of the popular MapReduce model to handle iterative tasks. Although MapReduce is proven to be widely efficient in processing batch-mode tasks, it performs poorly on applications that reuse the same working data on several parallel operations. This is because, with MapReduce, each task runs as a separate job and hence has to read and write data from and to the disk. Hence, iterative jobs like graph query processing incur serious latency due to increased I/O usage.

Spark, on the other hand introduced the Resilient Distributed Dataset (RDD) abstraction of the data, through which data, once loaded can stay in memory until the application exits. The RDDs represent a read-only collection of objects that are partitioned across a given number of machines. Intermediate results generated can also be persisted in memory in the form of RDDs to be reused later. Spark also has a excellent dependency management that traces the lineage of RDDs and prevents unnecessary re-computations. These features make Spark an ideal framework for processing iterative jobs.

The potential for Spark in processing iterative graph operations has already been established. GraphX [71] is a graph processing framework developed on top of Spark with efficient implementations for common graph operations like PageRank and Connected Components. However, there is no known implementation that leverages the benefits of Spark to process graph path queries.

In this section, we introduce an iterative graph path querying framework utilizing the Apache Spark framework, that eliminates the need to construct heavy graph-level indexes. We also introduce a novel graph storage model that is memory and computation efficient to work in a distributed setting.

6.2 Proposed Solution

In our approach, we propose a new querying model that eliminates driver-based aggregations and data shuffles from joins, by utilizing the iterative benefits of Spark. We follow a "bi-directional structural" filtering mechanism, to prune the path at each level.

6.2.1 Data Model

The most discerning components for querying a labeled graph are the edge and vertex labels that identify the neighborhood of a vertex. By storing the graph data in a $< s, p, o >$ format, which is more popular with RDF data, the collective neighborhood information of the vertex is lost. Splitting the graph into its vertex and edge components, again, does not offer a unified representation of the neighborhood. An adjacency list format, on the other hand, successfully collects the neighborhood of the vertex. In our implementation, we use a modified adjacency list format, that stores the neighborhood of a vertex, sorted in the order of the edge labels. Though a property graph has both edge and vertex labels, we chose to organize the neighborhood by the edge labels, as they form the first point of connection between two vertices. For those graphs without an edge label, our implementation can be easily modified to sort the neighborhood by vertex type.

Each vertex is identified by a unique id. We do not consider edges as first class objects, since our aim is to assimilate the neighborhood information into one cohesive structure. The graph structure is stored as an RDD of the form

```
RDD[(Int,Byte,Map[(Byte,Byte),Array[Int]])]
```

Here, the first `'Int'` value gives the vertex id of the vertex, the next `'Byte'` value gives the vertex type and the `Map` object gives the neighborhood of the vertex, sorted by edge type and direction. The `(Byte,Byte)` serves as the key to the map object, where the first `'Byte'` refers to the edge type and the second `'Byte'` refers to the direction indicated by `'o'` for outgoing and `'i'` for incoming. The value for each map key is an array of neighbor vertex ids that belongs to the given edge type and direction. An example vertex record for *Vertex*

6 of Figure 1.3(a) is given below:

```
( 6,
 ‘A’,
  Map((‘a’,‘o’)->Array(8,9),
          (‘b’,‘i’)->Array(7),
          (‘d’,‘i’)->Array(5)
         )
)
```

Where, the 6 represents the vertex id, *‘A’* represents the vertex type of *Vertex 6*. Now, looking at the neighborhood of *Vertex 6*, we see that it has two outgoing edges of type *‘a’* connecting V*ertices 8 and 9*, one incoming edge of type *‘b’* from *Vertex 7* and one incoming edge of type *‘d’* from *Vertex 5*. Though we have currently not included properties for edges and vertices, they can be easily incorporated in the storage model in a Map format.

Each vertex contains such a record and the graph is represented by the vertex records of all the vertices, stored as an RDD. Let us call this `graphRDD`. Once loaded and partitioned, this structure can be used to process a number of queries without having to re-read from the disk. Also, Spark's intelligent lineage handling prevents unnecessary re-calculations by keeping track of the descent tree for each RDD. We distribute the vertex information across the cluster by hashing on the vertex id. Since the vertex ids are continuous numbers, this gives us fairly balanced partition.

6.2.2 Query Processing

Building indexes for property graphs is both space and time expensive as we would need an index structure to represent each property type used. If we were to also index the graph structure, it becomes even more prohibitive as there does not exist a single compact structural index that can comprehensively answer a variety of graph queries. Our model does not require building indexes for the whole graph, and instead makes a single scan

across the graph to retrieve the set of likely candidates for each segment of the query. These vertices, the `'prune set'`, will be used to grow the path further and prune non-viable paths.

Prune Set:

The prune set is closely tied with the structure of the query. The query is divided into individual blocks and the prune set contains the set of vertices that satisfy the query requirements for each block. Let us first look into the structure of the query. We divide a query of length n into three different segments as follows:

1. 1 head segment of the form (VL, EL, ED)
2. n-2 middle segments of the form
   (LEL, LED, VL, REL, RED)
3. 1 tail segment of the form (EL, ED, VL)

where, VL denotes the vertex label, EL denotes the edge label, ED, the edge direction, LEL the left edge label, LED the left edge direction, REL the right edge label and RED the right edge direction. For example, the query in Figure 1.3(b) can be divided into 4 segments as follows:

```
Segment 0  - (A,o,a)
Segment 1  - (a,i,B,o,c)
Segment 2  - (c,i,D,o,a)
Segment 3  - (a,i,C)
```

The querying process begins by retrieving the prune set, i.e., the set of vertex ids that satisfy the structural requirements for each query segment. The head and tail segments look for vertices that match the vertex label VL and also has at least one edge of type EL in the direction ED. The middle segments however take advantage of the bi-directional information available in the neighborhood map and filter out only those vertices that match the label VL and have at least one edge of type LEL in direction LED and at least one edge of type REL in direction RED. The vertices thus identified are returned in the format of a

key-value pair identified by their matching query segment. Therefore, in the pre-querying phase, each segment of the query is associated with a set of vertex ids, the prune set, that satisfy the structural requirement of that section.

Let us look at an example for the network graph G and query graph Q given in Figure 1.3. The head segment of Q specifies that the vertex label should be A and should be connected to an outgoing edge of type a. The only vertices that satisfy this structural requirement are Vertices 3 and 6. Accordingly, these vertices are returned as matching sets for the head segment (segment-0) in the form `[(0,3),(0,6)]`. Similarly, segment 1 requires the vertex to be of label B, with one incoming edge of type a and one outgoing edge of type c. Clearly, only *Vertex 6* matches this and hence the result for segment 1 will be returned as [(1,6)]. If a vertex satisfies more than one segment of the query, each match is returned as a key-value pair identified by the query segment.

The prune set is stored as an `RDD` of form `RDD[(Int,Int)]`, with the first `'Int'` representing the query segment and the second `'Int'` representing the vertex id. We avoid grouping the vertex ids by the query segment to prevent an extra shuffle.

Path Growth:

Once the prune set is computed, the querying process repeats itself iteratively for each query segment as follows:

Step1: Collect the prune set for query segment `i`. Let us call this `vList`. Every vertex in `vList` satisfies query segment `i` and hence has at least one neighbor that matches the edge type of query segment `i`. Distribute `vList` among the partitions using a hash partitioning on the vertex ids.

Step2: For each vertex v in the prune set, retrieve the neighbors of v, $N_i(v)$, that match the edge type of query segment `i` from `graphRDD`. Since both the prune set and the `graphRDD` are distributed using the same hashing mechanism, they are co-located in the same partition and hence no additional data shuffles are required.

Step3: The vertices in $N_i(v)$ form the next level of vertices that need to be examined. How-

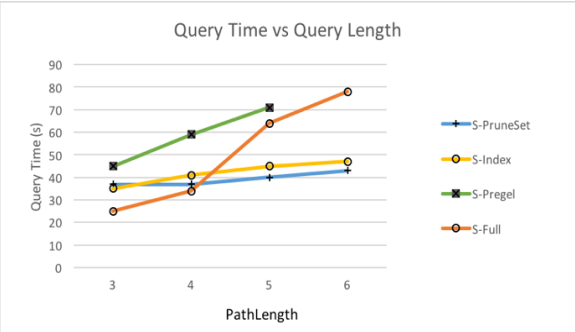


Figure (6.1) Query Time vs Query Length

ever, instead of using all the vertices in $N_i(V)$ for the next round of processing, we retrieve the prune set for query segment `i+1` to restrict $N_i(V)$ to only those vertices that satisfy the structural requirements for query segment `i+1`. Now the pruned neighbors $NP_i(V)$ becomes the next `vList`

Step4: Repeat the process from Step 2, for the new `vList`

This process continues until the end of the query is reached or until `vList` is empty. The advantage with this method is that, the larger portion of the data, namely the `graphRDD` is never shuffled throughout the querying process. The intermediate results obtained are collected and shuffled across the partitions to match the vertices with the respective `graphRDD` partitions. The prune set helps reduce this intermediate result and thereby further reduces the total amount of data transferred across the cluster. Also, by generating the prune set at the start of query execution, we perform one data scan to obtain the seed vertices for all the query segments.

6.3 Experimental Evaluation

We implemented our method in Apache Spark version 1.5.1 using scala. Experiments were performed on a 30-node cluster with commodity hardware with 4gb driver memory and 4gb executor memory. We used a hash partitioning strategy to partition the vertices. For the dataset, we used the LiveJournal dataset [72] obtained from the SNAP project [12] with

4.8M Nodes and 68.9M edges. The dataset is directed, but unlabeled. So, we randomly assigned vertex and edge labels. For this experiment, we used 8 edge and 8 vertex labels. We also randomly generated path queries of size ranging from 3 to 6.

6.3.1 Compared Methods

The motivation behind this work is to demonstrate the usage of Spark in processing graph path queries. Spark is known to provide 10x performance gains over MapReduce on iterative jobs due to eliminated disk reads and writes. Hence, we do not compare our method with existing MapReduce implementations, as the two operate on different paradigms. Instead, we compare our implementation against several Spark-based variations. The different implementations tested are listed down as follows:

1. S-PruneSet - This is our primary implementation described in Section 6.2

2. S-Index - This is a Spark based implementation wherein we created indexes for the graph structure. We created a separate index structure that organized the vertex ids based on the vertex type, and edge type. The structure of the index was of the form `RDD[(VL,ED,EL),Int]`, where `'VL'` denotes the vertex label, `'ED'` denotes the edge direction and `'EL'`, the edge label and `'Int'` denotes the vertex id. While the Prune Set is specific to the query and has to be calculated for each query, the index is created once for the graph and used for all the queries executed on it. This index structure is used similar to Prune Set to restrict the vertex ids for each query segment.

3. S-Full - This Spark implementation uses the entire graph and does not use either a prune set or an index.

4. S-Pregel - Spark comes with a graph processing library named GraphX, that provides an implementation of the popular Pregel model. We implemented a Pregel based path query model and used it as a test implementation.

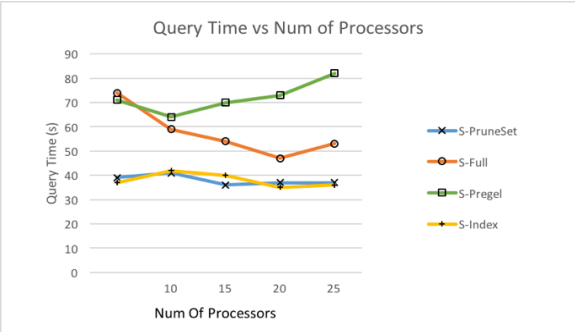


Figure (6.2) Query Time vs Num of Processors

6.3.2 Query Response Time

Figure 6.1 gives the query response time when running path queries of lengths ranging from 3 to 6. The experiments show an average of 20 queries instantiated on each implementation. We see that the S-PruneSet method outperforms all other implementations in terms of running time. the S-Pregel provided the slowest response and could not process path queries after length 6 due to memory limitations. This is because, the Pregel implementation of GraphX is based on an edge-based storage model, resembling a triple structure. Therefore, computation is defined in terms of the number of edges rather than on the number of vertices. Since, in a typical real world graph, the number of edges is many times larger than the number of vertices, this results in increased computation and data movement.The S-Index version comes a very close second to the S-PruneSet method. This is due to the fact that the S-PruneSet prunes the path using a bidirectional edge structuring, whereas the S-Index method considers only the uni-directional edge of the vertex. This also shows that Prune Set generation time is very negligible and has very little overhead in repeating it for each query. Notice that the S-Full method starts off being the fastest, but shows a very sharp rise in processing time at query length 5. This is because, with no pruning in place, the number of paths grows exponentially with increasing length. So, for shorter paths, the prune set/index generation time outweight the processing time, whereas, as the path length grows, this results in a significant raise. The S-PruneSet and S-Index methods on the other hand have a very linear increase in querying time.

We also observed the response time by varying the number of processors to measure the computation to communication ratio. The holy grail of distributed processing is to main the balance between distributing the processing and transferring data to and forth across the cluster. We noticed an interesting trend in the plot for implementations S-Full and S-Pregel. While the processing time reduced with increasing the processors, the trend started to reverse while continuing to increase the number of processors beyond a threshold for the same payload. This is because, with more number of processors, data gets split into smaller chunks, resulting in an increased communication overhead. S-PruneSet and S-Index showed very little variation to the number of processors. This is because the total amount of data processed is pruned at each stage and hence the amount of data transferred is small to cause an impact on the speedup factor. Figure 6.2 shows response time against number of processors using a query of length 5.

6.4 Conclusion

In this section, we demonstrated the use of Spark for iterative graph path queries. The RDD abstraction of Spark opens new possibilities by providing a persistent storage platform for repeated processing of data. Given the proven performance gains of Spark over MapReduce, to the best of our knowledge, there was yet no Spark based implementation for graph path query processing. We also showed several variations for path query processing using Spark.

PART 7

GRAPH TOPOLOGY ABSTRACTION FOR DISTRIBUTED PATH QUERIES

Querying graph data often involves identifying matching paths, either as an end product, or as an intermediate step for further graph analysis. Distributed graph querying, suffers from high communication to computation costs, due to challenges in constructing comprehensive structural indexes. This could result in severe performance degradation in terms of turnaround time, which often worsens with increasing graph size and density. In this section, we propose a novel topology abstraction layer, that helps improve query response time by reducing the communication overhead for selective exploration of large distributed graphs. We demonstrate the effectiveness of our model and also go on to show that our abstraction layer works well in both data-parallel and graph-parallel paradigms.

7.1 Introduction

Recent years have witnessed a rapid proliferation of sophisticated networks and a growing emphasis on inferring related properties. Graphs are ideal for representing relationships and, consequently, are used as the primary data structures in various fields such as Social Networks, Machine Learning, Semantic Web, Bio-informatics, Internet of Things etc.,. There is a wealth of information that could be mined from examining relationship structures between disparate entities. Adopting a property graph model, where vertices and edges can belong to specific types and can have a range of properties, caters well to describing unstructured and semi-structured data.

However, due to the absence of an optimal partitioning, distributed graph querying suffers from a high communication (T_{comm}) to computation (T_{comp}) cost. Analytical queries like PageRank [73], Connected Components [74] etc.,. are batch-mode queries and generally

involve accessing and processing the entire graph. These queries result in actions and modifications of the graph that are carried over in an iterative manner, i.e., the actions of step i directly affect the actions of step $i+1$. For such iterative batch processing, the communication to computation ratio is balanced since each communication step contributes towards the final result.

On the other hand, path queries are more selective in nature, where, the focus is on retrieving a subset of the graph that matches the query pattern. Most of the processing is dedicated towards locating the actual working set - a very small section of the graph - by pruning the non-matching components - which are generally much larger. So, not every communication contributes towards the result closure and much of the time is wasted in processing and shipping potentially inconsequential information across the cluster. In this section, we focus on improving the response time of interactive path queries in large labeled distributed graphs.

Significant improvements can be made in the response time of path queries, if the pruning of unrelated vertices could be done much ahead of time so that the payload for each round of processing is kept at a minimum. However, the unique nature of graphs, which inherently implicates tight coupling between the data and the structure, makes this difficult in a distributed environment. When a graph is partitioned, the neighborhood information gets distributed across various computing cores and hence, pruning of unrelated data is often not possible until a given vertex is accessed. In real time graphs, this causes increased data movement to determine connectivity and hence results in increased query latency.

In this section, we propose a novel structural abstraction layer, that helps create a topological coverage which could be used to prune unnecessary paths in distributed graph queries. We also introduce a distributed graph querying model based on Apache Spark[58] that eliminates data movement for distributed exploratory graph queries. The querying paradigm can be extended to any graph operation that includes selective querying - like regular path queries, structural queries and pattern matching queries. The abstraction layer is kept as light weight as possible and the entire structure is made locally available to

all the processors through broadcasting. With global information about the entire graph available locally, pruning can be done by the individual processors with a small amount of transportation.

7.2 K-Closure

Given a network G distributed over a cluster, and a query Q of length n, we define the k-closure of the vertex v_i encountered at length $i + k <= n$ of the query as the set $\{VL_k, EL_k\}$, where $VL_j = s_{v_k}$ and $EL_j = l_{v_k}$, $\forall v_k \in V(G)$, that are k hops from v_i. The $k - closure$ gives information about the vertex label and the edge labels of all the vertices that are k hops from it.

EXAMPLE 1 *Referring to Figure 1.3(a), the 1-closure for vertex 6, would include the the vertex labels and edge labels for vertices 8, 9, 7 and 5.*

In that regard, we define k-closure of a vertex as all the information that is required to determine if the vertex would contribute towards continuing the query for k more lengths.

7.3 Proposed Solution

In any distributed model, the graph itself is split and saved in a disjoint fashion among the processors, leaving no comprehensive detail about the graph that can help extend the visibility of a vertex beyond its local data. For batch mode queries like the PageRank, Connected Components etc.,. this does not cause a severe disadvantage since there is no decision making that goes beyond the localized vertex data at each stage. The working set comprises of the entire graph, and the only bottle neck is to transfer the changes from the one stage to the next.

However, for online graph queries, much of the processing involves deciding whether a vertex is going to contribute to the extension of the current path. While querying any decent sized graph, the working set, comprising of the number of potential matching paths, can grow exponentially with the size of the query. Effective pruning of non-viable paths,

early on in the querying framework, helps remove the communication bottleneck, as well as the processing time required to ship and compute the unfeasible paths across processors.

A global availability of the $k - closure$ of a vertex extends the visibility of the vertex contributing to the path well beyond its local data. This helps evaluate if a neighbor, k hops away, is going to satisfy the path k lengths from now, and help prune the path from growing further on. With a larger value of k, the time and space complexity of computing the $k - closure$ of a vertex becomes huge. A smaller k would provide the benefits of a look-ahead coverage, without impacting the storage and processing costs. The rest of the section describes our proposed solution that aims to promote early pruning of non-matching paths by making use of a topology abstraction layer that defines the $1 - closure$ of each vertex.

The increased difficulty in distributed graph processing can be attributed towards a lack of comprehensive indexing schemes to address a variety of querying patterns. Graph structural indexes are hard to build and are both memory and computation expensive operations. Typically, distributed graph path queries, follow either a partition based approach or a cloud based approach [54]. In the partition based approach, the query is partitioned into individual chunks tailored for each partition [49][50]. In the cloud based approach, the output pattern if formed by a series of join operations [75] [76]. The former methods rely on driver-based aggregations and hence suffer from performance bottlenecks while the latter use extensive joins to reach the output, thereby increasing data movement across the cluster.

In our approach, we propose a new querying model that eliminates driver-based aggregations as well as data shuffles from joins by making constructing the $1 - closure$ for each vertex. We adopt a simple iterative Breadth First Search pattern that builds the matching paths one query level at a time. In the first step, all vertices conforming to the first query block q_0 - the vertex label s_0 and edge label l_0 in direction d_0 - will be identified as seed vertices. In the subsequent steps, the graph is traversed along the edges and vertices that satisfy the query structure until query completion is achieved or there are no vertices/edges left to traverse. Any traversal that encounters a non-conforming vertex/edge is abandoned.

In our proposed model, we split the graph into two constituent regions - the Topology

Abstraction layer and the Neighborhood layer.

7.3.1 Topology Abstraction Layer

We propose the Topology Abstraction Layer layer to serve as the structural index for handling path queries in large distributed graphs. The abstraction layer, combined with the vertex neighborhood offers a compact representation of the $1-closure$ of each vertex and mimics the human thought process in answering pattern matching queries. The $1-closure$ of a vertex offers a 1-hop insight into the structure of a graph, since it represents the edge and vertex labels of all the immediate neighbors of the vertex. We expand on the concept of $1-closure$ to design a **Topolology Abstraction** for the graph, that can act as a look-ahead structural representation and help prune nonviable paths early on. The Topology Abstraction structure follows the same pattern as given in Figure 4.1. The first byte specifies if the vertex is deleted or not. This can be useful in online graph databases with active updates and deletions. The next $log(\theta)$ bytes represent the vertex label, where θ refers to the number of vertex types. The following $4*\omega$ bytes indicate the cardinality of the incoming and outgoing edges with two bytes each, arranged in increasing order of edge labels, where ω refers to the number of edge types used in the graph. An example Topology Abstraction record for vertex 6 in Figure 1.3(a), with $\theta = 4$ and $\omega = 4$ is shown in Figure 7.1. The reason for choosing a $1-closure$ instead of a larger number was to reduce preprocessing costs. A $0-closure$ would represent the vertex type and edge labels for the current vertex, while the $1-closure$ gives the structural information of all its immediate neighbors. While each record in the Topology Abstraction layer gives the $0-closure$ of a vertex, making this abstraction layer global helps define the $1-closure$ of the vertex in conjunction with the neighborhood record that is explained in the following section. So, essentially we reuse existing information and create an abstraction layer without additional processing or storage overhead. SPath[77] proposed to include the k-distance shortest paths for each vertex as the index for answering graph path queries. But, constructing such extensive indices would be computationally taxing and also reduce memory efficiency. A 1-closure, on the other hand,

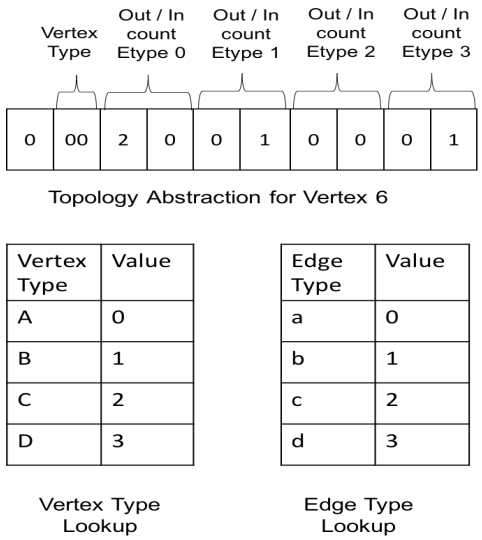


Figure (7.1) Topology Abstraction Record for Vertex 6

is easy to calculate and can be done during graph ingestion with $O(\Delta)$ computations for each vertex, Δ being the degree of the graph. However, the computation time and storage costs become exponential for any value of $k > 1$.

The storage structure is also simplified by using a $1 - closure$, whereas, a larger k value would require a two-dimensional structure like a HashMap to store the structural information. For ease of serialization/de-serialization, this layer is represented as a constant-sized byte array consisting of $1 + log(\theta) + (4 * \omega)$ bytes each. So, for $\theta = 8$ and $\omega = 8$, each record occupies 34 bytes of data. This layer grows with the vertex size of the graph and is agnostic of the edge size, as the Topology Abstraction layer only stores the cardinality of the vertex, not the entire neighborhood list. So, the total size of the Topology Abstraction layer $\propto |V|$ and is independent of graph density. Due to the compact representation, the Topology Abstraction layer has a very small memory footprint and fits in memory for most small-midsized graphs.

7.3.2 Neighborhood Layer

For the neighborhood record, we again choose a simplified model of an integer array containing the vertex ids of the neighboring vertices. The ordering of the vertices is main-

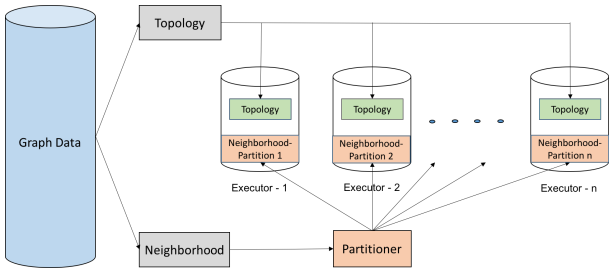


Figure (7.2) Data Distribution Strategy

tained in correlation with the ordering of edges given in the topology structure. This way, the topology abstraction layer serves as a decoder for the neighborhood structure and there is no need of a separate indicator to differentiate between neighbor types. This also facilitates easy access to neighbors connected by a particular edge type without having to iterate over the entire array. Having a map-based structure, that organizes the neighboring vertices by edge type would also work. But, it would require additional bytes to store the edge types $(2 * \omega)$ and would also lead to increased time in serializing it. A sample neighbor record for vertex 6 of graph G given in Figure **??** is shown in Figure 7.3. It has two outgoing neighbors $(vertex8, vertex9)$ connected by edge type 0, one incoming neighbor $(vertex7)$ connected by edge type 1, no incoming/outgoing neighbors of edge type 2 and 1 incoming neighbor $(vertex5)$ of edge type 3. The topology record for *Vertex 6*, given in Figure 7.1 is required to decode this information. The neighborhood layer is partitioned among the processors such that each processor stores the neighborhood records of a subset of vertices.

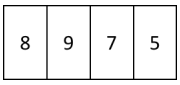


Figure (7.3) Neighborhood Record for Vertex 6

7.3.3 Data Parallel Implementation

Our distributed implementation splits the graph into two structures - the Topology - which encodes the Topology abstraction layer - and the Neighborhood - which contains the actual neighborhood of the graph. We propose to make the topology structure globally

available to all the processors across the cluster. This way, combined with the neighborhood record, each vertex has visibility on the neighborhood signature of the next level vertices and hence non-matching paths can be pruned without having to physically access the partition containing the next neighbor vertex. The advantages are two fold. Firstly, it eliminates an entire round of iteration, since the n^{th} iteration retrieves the results of stage $n+1$ of the query. Secondly, it reduces the number of messages passed in each round. In the absence of the Topology Abstraction, all neighbors of a vertex v_i that are connected by edge label l_i will be included for processing in the $i+1^{th}$ stage. The $1-closure$ of the vertex on the other hand helps examine the neighborhood structure and the vertex label of the neighboring vertex to determine if it would contribute to query continuation. If the neighbor vertex v_{i+1} of vertex v_i is found to be non-conforming to query block q_{i+1}, this path can be stopped in stage i instead of proceeding to stage $i+1$. For large dense graphs with high degree of neighbors, this greatly reduces the amount of data shipped across the cluster.

Listing 7.1: Vertex Filter Function

```
def filterVertex(vid:Int, qi:QueryItem) = {
   get topoTopIndex:Array[Byte] = broadCastTopology.get(vid).get
  if( vTop(1)==qi.vertexLabel && // vertex label check
     vTop(qi.edgeLabel) > 0) // edge structure check
    true
  else
    false
}
```

EXAMPLE 2 *Referring to Figure 1.3, vertex 3 satisfies q_0 since it is of type A and has an outgoing vertex of type B (Vertex 5). The next block, q_1, requires the vertex to have a label "B" and have an outgoing edge with label "c", which is clearly not satisfied by Vertex 5. Examining the 1-closure of Vertex 3 would have indicated that Vertex 5 does not lead to query continuation and hence would have not included Vertex 3 as a possible candidate. So,*

instead of physically accessing Vertex 5 to determine its inclusion in the path, this decision can be made one step ahead at Vertex 3.

The Topology Abstraction layer can be pre-computed for the graph and hence does not affect response time during actual querying. Also, the Topology Abstraction and Neighborhood structures together define the graph. Hence there is no additional storage cost for the $1 - closure$ representation. The flow of the query execution is given as follows:

Step1: Broadcast the *topology abstraction* across the cluster. It is of constant size and is designed to be compact to reduce the memory footprint.
Step2: Partition the *neighborhood structure* using a partitioning scheme. We persist the partitioning to avoid re-computation for further rounds.
Step3: Select the seed vertices, sv_0, conforming to the 0_{th} query block. This step is done in the master node and forms the 0^{th} iteration of the query.
Step4: Partition the seed vertices sv_0 using the same partitioning strategy used for partitioning the topology structure so that they are in the same partition. Now each partition has access to three critical regions

1. The vertex list v_i that passed the previous query round q_i. These vertices are guaranteed to extend the query at least by one more edge.

2. The neighbor vertices of each v_i. This information is obtained from the *neighborhood structure* of the vertex

3. The topology abstraction information for the next level neighbors v_{i+1} of each v_i, which is available through the broadcasted topology structure array in conjunction with the *neighborhood structure*

Step5: Retrieve the next level neighborhood vertices v_{i+1} for each vertex v_i
Step6: Using the topology abstraction information, filter vertices from v_{i+1} that satisfy query block q_{i+1}. These become the seed vertices sv_{i+1}, for block q_{i+1}

Step7: Send filtered vertices sv_{i+1} to STEP 4 for the next round of processing

Listing 7.2: Path Query for Generic Data-Parallel Paradigm

```
def PathQuery1(query:Query, bTop:BroadCast[Array[Byte]],
           tPart:RDD[Array[Int]]) = {
  //seed vertices with empty queue
  var vList = topology.filter(x=>filterVertex(x,query.head))
         .map((_,Queue())) // in driver
  // parallel computation starts
  for(i=1; i< query.size-1 && !vList.isEmpty, i++){
    val currentIndex = i
    def getNextVList(vList:Iterator[(Int,Queue[Int])],
       vTop:Iterator[(Int,Array[Int])])= {
      val vMap = vTop.toMap
      vList.map(x=>{
        vMap  .get(x._1) //get neighborhood
        //filter next query level
        .filter(x=>filverVertex(x,query.get(i+1)))
        .flatMap((_,x._2.push(x._1))) //add to queue
      })
    }
    //get next level vertices (done in parallel)
    vList = sc.parallelize(vList)
    .join(tPart)(getNextVlist)
  }
}
```

Since the seed vertices and the neighborhood records are partitioned using the same strategy, they are co-located in the same partition. This eliminates shuffles during the following

"join" operation. The topology and neighborhood structures are persisted in memory for the duration of the query. The only data movement in the entire algorithm is from partitioning the result from each iteration. The Topology Abstraction layer helps reduce the result set size in each iteration, thereby directly contributing to improved query performance. The $filterVertex$ function given in Listing 7.1 checks the neighborhood of each vertex to determine if it satisfies the vertex and edge types of the query. This function is called in STEP 6 of the query evaluation process. Note that this function checks the neighborhood structure of the next level vertices by utilizing the Topology Abstraction layer. The actual algorithm is given in Listing 7.2. Note that the algorithms follow a scala style syntax.

7.3.4 Graph-Parallel Implementation

We show that our Topology Abstraction layer can also be used in a graph-parallel paradigm. We implemented our Topology Abstraction layer using a Pregel-based framweork for graph computation. Pregel[55] works on the model of individual vertex mutations that get passed on to the neighboring vertices in a sequence of supersteps.

The path information is propagated across the graph in the form of a queue. Each vertex is initialized with an empty queue. When a vertex receives a message from any of its neighbors, it checks the neighborhood of its next level neighbors (through the Topology Abstraction layer) to determine if any of them satisfy the query predicate. If a matching neighbor vertex is found, it adds itself to the message queue and propagates the message to all the satisfiying neighbors. Those neighboring vertices that received a message in the previous round now check the neighborhood of their next level vertices, and upon finding a conforming neighbor, add themselves to the queue and propagate the message to all the conforming neighbors. This process continues until the query is fulfilled or until there are no active vertices that can contribute to the query. Keeping in structure with Pregel, we define four elements - the initial message, vertex program, send message and gather message. Listing 37.3 gives the scala algorithm for the pregel based implementation.

Listing 7.3: Path Query for Pregel Implementation

```
def PathQuery2(g:Graph[V,E], query:Query) = {
  g = g.subgraph(vpred=id=> filterVertex(id,query.head))//filter
     .mapV(v=>(List(Queue()),0)) //initialize
  initMsg = (List(Queue()),0)
  def vProg(v:Id, m:List[Queue[Id]]) = {
    //add itself to path
    val newMsg = m.foreach(_._1.push(v))
    return(newMsg,m._2+1)//increase query index
  }
  def sendMsg(t:Triplet)={
    val queryIndex = t.src.msg._2
    if(queryIndex<query.size && filterVertex(t.destId,query.get(queryIndex)))
           t.src.msg //send msg
    else
      None //no message sent
  }
  def gatherMsg(a:List[Queue[Id]], b:List[Queue[Id]]) =
    a ++ b //concatenate all msgs

  return Pregel(g,vProg,initMsg,sendMsg,gatherMsg)
}
```

7.4 Experimental Evaluation

We implemented both of the above mentioned algorithms using the Spark framework in scala. Experiments were performed on a 30-node cluster with commodity hardware with 4gb driver memory and 2gb executor memory. We used a hash partitioning strategy to partition the vertices. For the dataset, we used the LiveJournal dataset [72] obtained from the SNAP project [12] with 4.8M Nodes and 68.9M edges. The dataset is directed, but unlabeled. So,

we randomly assigned vertex and edge labels. For this experiment, we used 8 edge and 8 vertex labels. We also randomly generated path queries of the type given in Figure **??**(b) with size ranging from 3 to 6.

7.4.1 Performance Analysis

The motivation behind this paper is to introduce the Topology Abstraction layer and the usage of Spark for graph processing in the distributed scenario. We are aware of several distributed path query processing methods like [78] [56]. However, our aim is to demonstrate the use of a Topology Abstraction layer in processing graph queries. We do not claim that our path query processing model outperforms existing ones, but merely suggest that introducing a globally available abstraction layer can help speedup path queries in a variety of distributed paradigms. The suggested abstraction framework can be easily tailored to fit in any selective graph exploration framework. Accordingly, we tested our abstraction layer in two different graph processing paradigms - the data-parallel and graph-parallel models.

For the data-parallel model, we used our distributed querying model described in Section 4.3 and for the graph-parallel model, we built our framework described in Section 4.4 over GraphX's Pregel implementation. GraphX is a graph parallel engine built on top of Spark to enable efficient computation of graph-specific algorithms. We do not compare our Spark implementation with existing MapReduce implementations since our primary aim is to showcase the benefits of a structural abstraction and hence only compare the implementations with the Topological Abstraction against those without it. Also, comparing it with MapReduce implementations would be unfair as MapReduce and Spark operate on different paradigms and Spark has been shown to produce a speedup of more than 10x when compared to MapReduce [58] due to its reduced dependency on I/O operations.

Our results show that adding a topological abstraction layer helps significantly improve query results. For the LiveJournal dataset, the total size of the Topology Abstraction layer was 163MB. The compact size of the Topology layer enabled us to store it in a broadcast fashion and use it as a lookup table to help prune non-feasible paths. We refer to

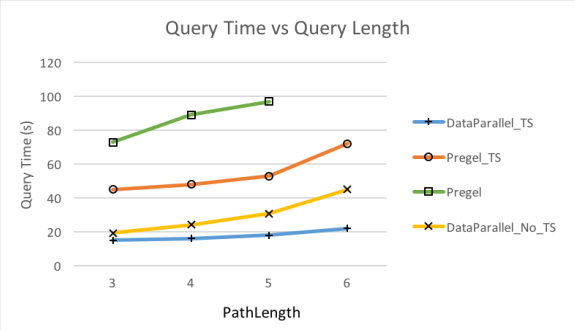


Figure (7.4) Query Time vs Path Length

the data parallel implementation with and without the Topology Abstraction layer as *DataParallel_TS* and *DataParallel_No_TS* respectively, whereas the corresponding graph parallel implementations are referred to as *Pregel_TS* and *Pregel* respectively.

Query Response Time Figure 7.4 gives the query response time when running path queries of lengths ranging from 3 to 6 using 10 processors. The experiments show an average of 20 queries instantiated on each implementation. We see that the Topology Abstraction reduces turnaround time in both the data-parallel and graph-parallel frameworks. This happens partly because the abstraction layer reduces the total computation time by reducing the set of vertices in each stage. However, the real benefit of the Topology Abstraction is in the reduction of communication time. The amount of nonviable data transfered across the cluster is greatly reduced leading to an improvement in response time.This is especially true in the case of the graph-parallel model, since one-on-one communication between vertices is heavier here than in our data-parallel case. As we varied the size of the query, we see that the difference in processing time increases substantially in the non-abstraction implementations, while it remains almost constant for the data-parallel case. Traditional GraphX, without the Topology Abstraction layer failed to return a result for a query size greater than 5 due to memory overhead issues with the executors.

We also observed the response time by varying the number of processors to measure the computation to communication ratio. The holy grail of distributed processing is to main the balance between distributing the processing and transferring data to and forth across the

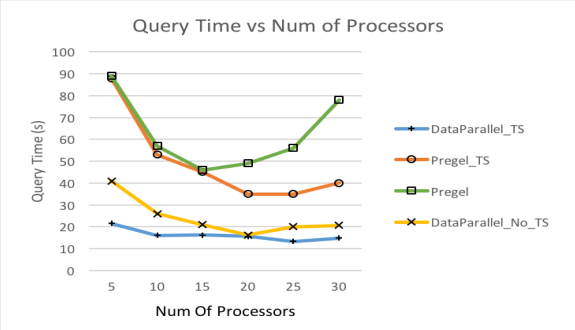


Figure (7.5) Query Time vs Num of Processors

cluster. We noticed an interesting trend in the plot for the non-TS implementations, given in Figure 7.5. While the processing time reduced with increasing the processors, the trend started to reverse while continuing to increase the number of processors beyond a threshold for the same payload. This is because, with more number of processors, data gets split into smaller chunks, resulting in an increased communication overhead. This threshold was higher for the TS implementations than the non-TS ones. The Pregel implementation with the TS layer showed a more obvious improvement over the non-TS Pregel version in sensitivity to increased parallelism. This is due to the fact that, with a higher number of partitions, the graph is split into more fine grained sections. So, there is far greater data movement across the cluster. Our abstraction layer helps reduce the amount of data transferred across the network by pruning paths early on.

Memory Characteristics We also measured the amount of memory consumed and shuffled across the cluster. Figure 7.6 shows the total input size, the read shuffle size and write shuffle size for each of the four methods. These experiments belong to a query of length 4 run using 10 executors. Note that the Y-axis denotes the data size in MB using the logarithmic scale. As expected, our Topology Abstraction layer helps reduce the total amount of data moved across the processors, in both the data-parallel and graph-parallel implementations. We also observed an interesting characteristic with the input size. The data-parallel implementation operated on a relatively smaller input size than the graph-parallel case. This is because, our data model used in the data-parallel method used a storage

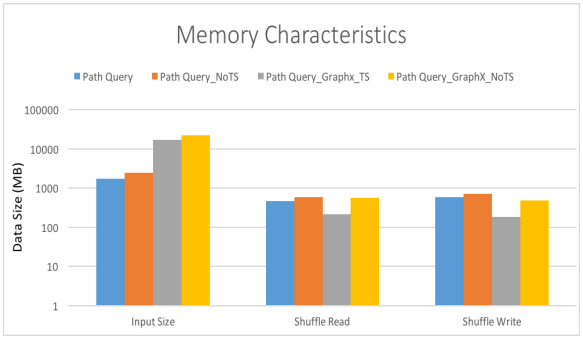


Figure (7.6) Memory Characteristics

format optimized for path queries, whereas the graph-parallel implementation operated on edge triples. Since the number of edges is many times larger than the vertex size for the given dataset, the data-parallel implementation shows reduced data sizes.

However, the shuffle size is lesser for the graph-parallel implementation. This is because, in the GraphX implementation, messages to the same vertex are aggregated into one structure, whereas in our data-parallel method, we treat each message individually. This is an area where further optimization could be done.

7.5 Conclusion

In this section, we presented a Topology Abstraction layer for graphs that can help speedup distributed graph query processing. We defined a compact structural layer that can comprehensively answer the 1-closure for a vertex. Making the abstraction layer globally available across the cluster creates a global lookup that can help prune non-matching paths early on. In this research, we aim to present a generalized data abstraction layer for graphs that could be used in multiple scenarios to improve response time. We also presented a Spark-based distributed model to implement path queries that used our Topological Abstraction layer. We also demonstrated that our Topological Abstraction layer can be used in both data-parallel and graph-parallel paradigms.

PART 8

CONCLUSIONS

At the end of this dissertation, we have developed a graph storage structure that can efficiently store and process labeled graphs. We also improved the scalability of our method by developing an adaptive caching framework that allows us to use larger graphs in a single processor environment. In addition, we also explored the usage of Spark for interactive graph queries. In doing so, we observe that adding a Topology Abstraction for the graph significantly reduced the querying speed for most path queries.

We identified two major kinds of big data applications possible with graphs. The first is to use graphs to represent connected entities and use them for point querying. Here, the operational model is largely random and the graph is explored in a selective format. For these kinds of applications, a NoSQL system, including ours, is the best way to go, since response time is a major factor. Since these involve frequent updates and modifications, a complete persistent structure that supports transactional access and a closer coupling between the physical and logical representation of the data would be beneficial. The other kinds of applications are analytical applications that are held offline. These kinds of operations involve applying complex graph operations on the entire graph. These applications look to leverage the connected nature of the data by inferring and mining information that is not evidently seen. For these kinds of applications, a system with large processing capability that can smoothly scale with the volume of data is essential. These kinds of systems do not have the persistence or transaction abilities required in a full fledged graph database. The first half of our dissertation focused on developing an online graph database that could be used in multi-disciplinary scenarios. The second half on the other hand tried to use a common analytics framework for a query processing scenario.

The biggest takeaway from this dissertation is the impact of the storage representation

format on performance. We studied several data storage formats including adjacency matrices, columnar representations, encoded bitmaps, linked list structures, etc., But we found that the storage format is closely tied with the graph application it is used for. Analytical tasks like PageRank, Connected Components etc., are mostly memory based operations. Due to the size of the data involved, for mid-large sized graphs, these operations have to performed in a distributed setting. This scenario takes away the dependencies from the data structure. However, for online graphs, the performance of the database is closely tied to the data structure used. We saw huge differences when the same operation used different storage formats, since we were able to leverage and adapt the physical representation to suit the application.

Though much of the dissertation has been focused on path queries, the techniques can be extended to a generalized motif matching problem with relatively few changes. We also observed that more complex techniques do not necessarily translate to improved performance, but rather adapting the technique to suit the basic representation of the data helped improve overall performance. Basic graph traversals outweigh the costs due to joins and hence simplified traversal operations can obtain better performance than other complex algorithms for graph exploration.

REFERENCES

[1] E. Protalinski. Facebook passes 1.44b monthly active users and 1.25b mobile users. http://venturebeat.com/2015/04/22/facebook-passes-1-44b-monthly-active-users-1-25b-mobile-users-and-936-million-daily-users/. [Online; accessed 04-September-2015].

[2] Web data commons - hyperlink graphs. http://webdatacommons.org/hyperlinkgraph. [Online; accessed 04-September-2015].

[3] F. Bonchi, G. De Francisci Morales, and M. Riondato, "Centrality measures on big graphs: Exact, approximated, and distributed algorithms," in *Proceedings of the 25th International Conference Companion on World Wide Web*, ser. WWW '16 Companion. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, pp. 1017–1020. [Online]. Available: http://dx.doi.org/10.1145/2872518.2891063

[4] V. Galluzzi, "Real time distributed community structure detection in dynamic networks," in *Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM International Conference on*, Aug 2012, pp. 1236–1241.

[5] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[6] L. Euler, "Solutio problematis ad geometriam situs pertinentis," *Commentarii academiae scientiarum Petropolitanae*, vol. 8, pp. 128–140, 1741. [Online]. Available: http://arxiv.org/abs/1006.4990

[7] A. Inokuchi, T. Washio, and H. Motoda, "Complete mining of frequent patterns from

graphs: Mining graph data," *Mach. Learn.*, vol. 50, no. 3, pp. 321–354, Mar. 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1021726221443

[8] J. Huan, W. Wang, J. Prins, and J. Yang, "Spin: Mining maximal frequent subgraphs from graph databases," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '04. New York, NY, USA: ACM, 2004, pp. 581–586. [Online]. Available: http://doi.acm.org/10.1145/1014052.1014123

[9] I. Dhillon, Y. Guan, and B. Kulis, "A fast kernel-based multilevel algorithm for graph clustering," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05. New York, NY, USA: ACM, 2005, pp. 629–634. [Online]. Available: http://doi.acm.org/10.1145/1081870.1081948

[10] S. B. Navathe and M. Ra, "Vertical partitioning for database design: A graphical algorithm," *SIGMOD Rec.*, vol. 18, no. 2, pp. 440–450, Jun. 1989. [Online]. Available: http://doi.acm.org/10.1145/66926.66966

[11] X. Yan, F. Zhu, P. S. Yu, and J. Han, "Feature-based similarity search in graph structures," *ACM Trans. Database Syst.*, vol. 31, no. 4, pp. 1418–1453, Dec. 2006. [Online]. Available: http://doi.acm.org/10.1145/1189769.1189777

[12] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, jun 2014.

[13] P. Boldi and S. Vigna, "The webgraph framework i: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 595–602. [Online]. Available: http://doi.acm.org/10.1145/988672.988752

[14] I. Al-Furaih and S. Ranka, "Memory hierarchy management for iterative graph structures," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the*

First Merged International ... and Symposium on Parallel and Distributed Processing 1998, Mar 1998, pp. 298–302.

[15] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. New York, NY, USA: ACM, 1969, pp. 157–172. [Online]. Available: http://doi.acm.org/10.1145/800195.805928

[16] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, ser. ICDM '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 721–. [Online]. Available: http://dl.acm.org/citation.cfm?id=844380.844811

[17] A. A.-B. D. C.-M. C. H. K. M. D.A. Bader, J. Berry and S. Poulos, "Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation," Georgia Institute of Technology, Tech. Rep., 2009.

[18] R. Giugno and D. Shasha, "Graphgrep: A fast and universal method for querying graphs," in *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 2, 2002, pp. 112–115 vol.2.

[19] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 16:1–16:45, Sep. 2009. [Online]. Available: http://doi.acm.org/10.1145/1567274.1567278

[20] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient rdf store over a relational database," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/2463676.2463718

[21] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura, "A path-based relational rdf database," in *Proceedings of the 16th Australasian Database Conference - Volume 39*, ser.

ADC '05. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2005, pp. 95–103. [Online]. Available: http://dl.acm.org/citation.cfm?id=1082222.1082233

[22] T. Neumann and G. Weikum, "Rdf-3x: A risc-style engine for rdf," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 647–659, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1453856.1453927

[23] O. Erling and I. Mikhailov, *RDF Support in the Virtuoso DBMS*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 7–24. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02184-8_2

[24] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale rdf data," *Proc. VLDB Endow.*, vol. 6, no. 4, pp. 265–276, Feb. 2013. [Online]. Available: http://dx.doi.org/10.14778/2535570.2488333

[25] M. Cai and M. Frank, "Rdfpeers: A scalable distributed rdf repository based on a structured peer-to-peer network," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 650–657. [Online]. Available: http://doi.acm.org/10.1145/988672.988760

[26] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[27] M. Vora, "Hadoop-hbase for large-scale data," in *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, vol. 1, Dec 2011, pp. 601–605.

[28] db4o object oriented database. http://www.db4o.com/. [Online; accessed 28-September-2014].

[29] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley db," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '99.

Berkeley, CA, USA: USENIX Association, 1999, pp. 43–43. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268708.1268751

[30] K. Chodorow and M. Dirolf, *MongoDB: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2010.

[31] Neo4j graph database. http://www.neo4j.org/. [Online; accessed 28-September-2014].

[32] Sparksee scalable high-performance graph database. http://www.sparsity-technologies.com/. [Online; accessed 28-September-2014].

[33] Orientdb document graph nosql dbms (database). http://www.orientechnologies.com/orientdb/. [Online; accessed 28-September-2014].

[34] S. Srinivasa, "Lwi and safari: A new index structure and query model for graph databases, comad 2005," in *In Proc. 11th International Conference on Management of Data (COMAD 2005*, 2005.

[35] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang, "Computing label-constraint reachability in graph databases," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 123–134. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807183

[36] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 335–346. [Online]. Available: http://doi.acm.org/10.1145/1007568.1007607

[37] H. He and A. K. Singh, "Closure-tree: An index structure for graph queries," in *22nd International Conference on Data Engineering (ICDE'06)*, April 2006, pp. 38–38.

[38] S. Zhang, M. Hu, and J. Yang, "Treepi: A novel graph indexing method," in *2007 IEEE 23rd International Conference on Data Engineering*, April 2007, pp. 966–975.

[39] A. Shokoufandeh, D. Macrini, S. Dickinson, K. Siddiqi, and S. W. Zucker, "Indexing hierarchical structures using graph spectra," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 7, pp. 1125–1140, July 2005.

[40] L. Zou, L. Chen, J. X. Yu, and Y. Lu, "A novel spectral coding in a large graph database," in *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '08. New York, NY, USA: ACM, 2008, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/1353343.1353369

[41] H. Jiang, H. Wang, P. S. Yu, and S. Zhou, "Gstring: A novel approach for efficient search in graph databases," in *2007 IEEE 23rd International Conference on Data Engineering*, April 2007, pp. 566–575.

[42] Titan distributed graph database. http://thinkaurelius.github.io/titan/. [Online; accessed 28-September-2014].

[43] Infinitegraph. http://www.objectivity.com/infinitegraph. [Online; accessed 28-September-2014].

[44] H. Choi, J. Son, Y. Cho, M. K. Sung, and Y. D. Chung, "Spider: A system for scalable, parallel / distributed evaluation of large-scale rdf data," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, ser. CIKM '09. New York, NY, USA: ACM, 2009, pp. 2087–2088. [Online]. Available: http://doi.acm.org/10.1145/1645953.1646315

[45] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "Gbase: An efficient analysis platform for large graphs," *The VLDB Journal*, vol. 21, no. 5, pp. 637–650, Oct. 2012. [Online]. Available: http://dx.doi.org/10.1007/s00778-012-0283-9

[46] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of SIGMOD 2013*. ACM SIGMOD, June 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=183710

[47] R. Chen, X. Weng, B. He, and M. Yang, "Large graph processing in the cloud," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 1123–1126. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807297

[48] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *CoRR*, vol. abs/1204.6078, 2012. [Online]. Available: http://arxiv.org/abs/1204.6078

[49] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL querying of large RDF graphs," *PVLDB*, vol. 4, no. 11, pp. 1123–1134, 2011. [Online]. Available: http://www.vldb.org/pvldb/vol4/p1123-huang.pdf

[50] K. Hose and R. Schenkel, "WARP: workload-aware replication and partitioning for RDF," in *Workshops Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, 2013, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/ICDEW.2013.6547414

[51] T. Milo and D. Suciu, "Index structures for path expressions," in *Proceedings of the 7th International Conference on Database Theory*, ser. ICDT '99. London, UK, UK: Springer-Verlag, 1999, pp. 277–295. [Online]. Available: http://dl.acm.org/citation.cfm?id=645503.656266

[52] L.-D. Tung, Q. Nguyen-Van, and Z. Hu, "Efficient query evaluation on distributed graphs with hadoop environment," in *SoICT*, December 2013.

[53] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu, "Adding regular expressions to graph reachability and pattern queries," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, April 2011, pp. 39–50.

[54] P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao, "Processing SPARQL queries over linked data-a distributed graph-based approach," *CoRR*, vol. abs/1411.6763, 2014. [Online]. Available: http://arxiv.org/abs/1411.6763

[55] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[56] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel, "Horton+: A distributed system for processing declarative reachability queries over partitioned graphs," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1918–1929, Sep. 2013. [Online]. Available: http://dx.doi.org/10.14778/2556549.2556573

[57] "Apache giraph," http://apache.giraph.org.

[58] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[59] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," in *In Parallel Object-Oriented Scientific Computing (POOSC*, 2005.

[60] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "Stapl: Standard template adaptive parallel library," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, ser. SYSTOR '10. New York, NY, USA: ACM, 2010, pp. 14:1–14:10. [Online]. Available: http://doi.acm.org/10.1145/1815695.1815713

[61] G. Karypis and V. Kumar, "Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0," Tech. Rep., 1995.

[62] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing—a

fast access method for dynamic files," *ACM Trans. Database Syst.*, vol. 4, no. 3, pp. 315–344, Sep. 1979. [Online]. Available: http://doi.acm.org/10.1145/320083.320092

[63] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, Jun. 1979. [Online]. Available: http://doi.acm.org/10.1145/356770.356776

[64] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84. New York, NY, USA: ACM, 1984, pp. 47–57. [Online]. Available: http://doi.acm.org/10.1145/602259.602266

[65] N. Lester, A. Moffat, and J. Zobel, "Efficient online index construction for text databases," *ACM Trans. Database Syst.*, vol. 33, no. 3, pp. 19:1–19:33, Sep. 2008. [Online]. Available: http://doi.acm.org/10.1145/1386118.1386125

[66] S. Sakr, S. Elnikety, and Y. He, "G-sparql: A hybrid engine for querying large attributed graphs," Tech. Rep. MSR-TR-2011-138, December 2011. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=157417

[67] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[68] N. Edmonds, J. Willock, T. Hoefler, and A. Lumsdaine, "Design of a Large-Scale Hybrid-Parallel Graph Library," in *International Conference on High Performance Computing, Student Research Symposium*. IEEE, Dec. 2010.

[69] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010. [Online]. Available: http://arxiv.org/abs/1006.4990

[70] J. Krishnamani and R. Sunderraman, "A high performance graph database for big data," in *Big Data & Analysis for Business*. Society for Educational & Research Development, 2015, pp. 27–36.

[71] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 599–613. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685096

[72] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *CoRR*, vol. abs/0810.1355, 2008. [Online]. Available: http://arxiv.org/abs/0810.1355

[73] F. Chung and W. Zhao, *Fete of Combinatorics and Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ch. PageRank and Random Walks on Graphs, pp. 43–62. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13580-4_3

[74] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii, "Connected components in mapreduce and beyond," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 18:1–18:13. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670997

[75] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham, "Heuristics-based query processing for large rdf graphs using cloud computing," *IEEE Trans. on Knowl. and Data Eng.*, vol. 23, no. 9, pp. 1312–1327, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2011.103

[76] K. Rohloff and R. E. Schantz, "High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store," in *Programming Support Innovations for Emerging Distributed Applications*, ser. PSI

EtA '10. New York, NY, USA: ACM, 2010, pp. 4:1–4:5. [Online]. Available: http://doi.acm.org/10.1145/1940747.1940751

[77] P. Zhao and J. Han, "On graph query optimization in large networks," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 340–351, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920887

[78] U. Kang and C. Faloutsos, "Big graph mining: Algorithms and discoveries," *SIGKDD Explor. Newsl.*, vol. 14, no. 2, pp. 29–36, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2481244.2481249